UD08 – FICHEROS. FLUJOS DE ENTRADA Y SALIDA. REGEX

Contenido

1 Ficheros y directorios 1
1.1 Clase Path 3
2 Directorios 5
1.1.1 Propiedades: 7
1.1.2 Métodos 7
3 Ficheros 9
4 Concepto de flujo 13
5 Flujos de entrada y salida de datos 17
6 Tareas comunes de E/S 30
7 Expresiones regulares. Patrones. 31
7.1 Editar cadenas 37

1 Ficheros y directorios

Un sistema de ficheros almacena y organiza los ficheros en algún soporte, como uno o más discos duros magnéticos o SSD, de manera que estos ficheros puedan ser fácilmente accesibles. Estos sistemas almacenan los ficheros en una estructura jerárquica en forma de árbol. En la base de la jerarquía encontramos uno o más nodos raíz. Bajo estos nodos raíz encontramos ficheros y directorios. Cada directorio puede contener ficheros y subdirectorios, los cuales pueden contener a su vez ficheros y subdirectorios y así sucesivamente, potencialmente sin límite de profundidad en los niveles de subdirectorios.

Una ruta (path) de acceso es una cadena que proporciona la ubicación de un archivo o un directorio. Una ruta de acceso no apunta necesariamente a una ubicación en el disco; por ejemplo, una ruta de acceso podría asignarse a una ubicación en memoria o en un dispositivo. El sistema operativo determina el formato exacto de una ruta de acceso.

Podemos identificar un fichero a través de su ruta (path) en el sistema de ficheros, empezando desde el nodo raíz. En Windows las rutas serán del estilo:

C:\Users\alumno\Documents

Y en Linux tienen el siguiente patrón:

/home/alumno/Documents

Una ruta puede ser relativa o absoluta. Una ruta absoluta incluye siempre desde el elemento raíz y la lista completa de directorios necesarios para encontrar el archivo, por ejemplo:

C:\Users\alumno\Documents\práctica1.docx

Las rutas de acceso relativas especifican una ubicación parcial: la ubicación actual se usa como punto de partida al buscar un archivo especificado con una ruta de acceso relativa. Una ruta relativa necesita la combinación con otra para acceder a un fichero.

Por ejemplo:

Documents\práctica1.odt

Sin más información, un programa no puede localizar de manera correcta el archivo en el sistema de ficheros. Para determinar el directorio actual, llamaremos al método

Directory.GetCurrentDirectory.

```
public class FicherosYDirectorios {
    public static void main(String[] args) {
        System.out.println("Clase de ejemplo para trabajar con rutas de acceso y directorios.");
        // Aquí puedes implementar ejemplos prácticos relacionados con los conceptos descritos.
    }
}
```

1.1 Clase Path

La clase estática Path ejecuta operaciones en instancias de String que contienen información de rutas de acceso de archivos o directorios. Estas operaciones se ejecutan de forma adecuada para múltiples plataformas.

El sistema operativo también determina el conjunto de caracteres que se usa para separar los elementos de una ruta de acceso y el conjunto de caracteres que no se pueden usar al especificar rutas de acceso. Debido a estas diferencias, los campos de la clase Path así como el comportamiento exacto de algunos de sus miembros dependerán de la plataforma.

Veamos unos ejemplos de uso:

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

public class TestPath {

   public static void testPath() throws IOException {
      Path path1 = Paths.get("C:/temp/MyTest.txt");
      Path path2 = Paths.get("C:/temp/MyTest");
      Path path3 = Paths.get("temp");

      // Comprobar si tiene extensión
      if (path1.toString().contains(".")) {
         System.out.println(path1 + " tiene una extensión.");
         System.out.println("PathRoot: " + path1.getRoot());
         System.out.println("DirectoryName: " + path1.getParent());
         System.out.println("FileName: " + path1.getFileName());
         String fileName = path1.getFileName().toString();
         System.out.println("FileNameWithoutExtension: " + fileName.substring(0, fileName.lastIndexOf('.')));
         System.out.println("Extension: " + fileName.substring(fileName.lastIndexOf('.')));
      }

      // Comprobar si no tiene extensión
      if (!path2.toString().contains(".")) {
         System.out.println("\n" + path2 + " no tiene una extensión.");
      }

      // Comprobar si no es una ruta raíz
      if (!path3.isAbsolute()) {
         System.out.println("\nLa cadena " + path3 + " no contiene información de raíz.");
      }

      // Obtener la ruta completa
      System.out.println("La ruta completa de " + path3 + " es " + path3.toAbsolutePath());

      // Ruta temporal
      System.out.println("\n" + System.getProperty("java.io.tmpdir") + " es la ubicación para archivos
temporales.");

      // Archivo temporal
      Path                                                tempFile                                                =
Files.createTempFile(FileSystems.getDefault().getPath(System.getProperty("java.io.tmpdir")),   "tempFile",
".tmp");
      System.out.println(tempFile + " es un archivo disponible para usar.");
   }

   public static void main(String[] args) {
      try {
         testPath();
      } catch (IOException e) {
         System.err.println("Error al manejar archivos: " + e.getMessage());
      }
   }
}
```

2 Directorios

Las dos clases principales para trabajar con directorios en Java son File y Path/Files. La primera es una clase estática para realizar operaciones sobre directorios. Los objetos de la segunda representan un directorio en concreto y ofrecen información sobre él.

La clase File expone métodos estáticos para crear, mover y enumerar archivos en directorios y subdirectorios.

Método	Descripción
mkdir() / mkdirs()	Para crear un directorio.
delete() / deleteOnExit()	Borra un directorio.
System.getProperty("user.dir")	Obtiene el directorio actual de una aplicación.
listFiles(File::isDirectory) o Files.newDirectoryStream()	Devuelve los nombres de los subdirectorios (con sus rutas de acceso) del directorio especificado.
File.getAbsoluteFile().getRoot() o File.getParentFile().getParent()	Devuelve la información del volumen, la información de raíz o ambas para la ruta de acceso especificada.
listFiles(File::isFile) o Files.list(Path)	Devuelve los nombres de archivo (con sus rutas de acceso) del directorio especificado.
File.getParent() o Path.getParent()	Recupera el directorio principal de la ruta especificada, incluidas tanto las rutas de acceso absolutas como las relativas.
renameTo()	Mueve un archivo o directorio y su contenido a una nueva ubicación.

1. Crear directorios:
- File: Usa mkdir() para crear un solo directorio y mkdirs() para crear directorios con subdirectorios faltantes.
- Files: Métodos estáticos como createDirectory() y createDirectories() proporcionan mayor flexibilidad.

2. Eliminar directorios:
- File.delete() elimina un archivo o un directorio vacío.
- Files.delete() elimina directorios y arroja una excepción si no están vacíos.

3. Obtener el directorio actual:
- Se puede obtener el directorio actual usando System.getProperty("user.dir").

4. Listar subdirectorios o archivos:
- File.listFiles() devuelve una lista de File que puede filtrarse para distinguir archivos y directorios.
- Files.newDirectoryStream() es una opción más moderna que permite iterar sobre el contenido de un directorio.

5. Mover directorios:
- File.renameTo(File) puede mover archivos o directorios.
- Files.move(Path, Path) permite opciones avanzadas como sobrescribir archivos.

Veamos con un ejemplo como usar la clase File:

```
import java.io.File;

public class DirectoryTest {
   public static void main(String[] args) {
      String sourceDirectory = "C:\\current";
      String archiveDirectory = "C:\\archive";

      try {
         File sourceDir = new File(sourceDirectory);
         File archiveDir = new File(archiveDirectory);

         // Obtener todos los archivos .txt en el directorio fuente
         File[] txtFiles = sourceDir.listFiles((dir, name) -> name.endsWith(".txt"));

         if (txtFiles != null) {
            for (File currentFile : txtFiles) {
               // Crear un nuevo destino para el archivo en el directorio de archivo
               File destFile = new File(archiveDir, currentFile.getName());

               // Mover el archivo al directorio de archivo
               if (currentFile.renameTo(destFile)) {
                  System.out.println("Archivo movido: " + currentFile.getName());
               } else {
                  System.out.println("No se pudo mover el archivo: " + currentFile.getName());
               }
            }
         } else {
            System.out.println("No se encontraron archivos .txt en el directorio fuente.");
         }
      } catch (Exception e) {
         System.out.println("Ocurrió un error: " + e.getMessage());
      }
   }
}
```

1.1.1 Propiedades:

1. **Parent**
   En Java, la clase `File` tiene el método `getParent()` para obtener el directorio principal.

```
File dir = new File("C:\\MyDir\\SubDir");
String parentDir = dir.getParent();
System.out.println("Parent directory: " + parentDir);
```

2. **FullName**
   En Java, se usa `getAbsolutePath()` para obtener la ruta completa del directorio.

```
File dir = new File("C:\\MyDir\\SubDir");
System.out.println("Full path: " + dir.getAbsolutePath());
```

3. **Name**
   El método `getName()` devuelve solo el nombre del directorio o archivo.

```
File dir = new File("C:\\MyDir\\SubDir");
System.out.println("Directory name: " + dir.getName());
```

4. **Exists**
   El método `exists()` verifica si un archivo o directorio existe.

```
File dir = new File("C:\\MyDir\\SubDir");
System.out.println("Exists: " + dir.exists());
```

5. **Root**
   Para obtener la raíz, puedes usar `getAbsolutePath()` y extraer la raíz del directorio.

```
File dir = new File("C:\\MyDir\\SubDir");
File root = dir.toPath().getRoot().toFile();
System.out.println("Root: " + root.getAbsolutePath());
```

1.1.2 Métodos

1. **Delete**
   En Java, se usa el método `delete()` de la clase `File`. Ten en cuenta que este método solo funciona si el directorio está vacío.

```
File dir = new File("C:\\MyDir\\SubDir");
if (dir.delete()) {
    System.out.println("Directory deleted");
} else {
    System.out.println("Directory not empty or couldn't be
deleted");
}
```

2. **Create**
   En Java, puedes usar `mkdir()` para crear un solo directorio o `mkdirs()` para crear un directorio y todos los necesarios en la jerarquía.

```
File dir = new File("C:\\MyDir\\SubDir");
if (dir.mkdirs()) {
    System.out.println("Directory created");
} else {
    System.out.println("Failed to create directory");
}
```

3. **GetDirectories**
   Para listar subdirectorios, usa `listFiles()` con un filtro para verificar si es un directorio.

```
File dir = new File("C:\\MyDir");
File[] subDirs = dir.listFiles(File::isDirectory);

if (subDirs != null) {
    for (File subDir : subDirs) {
        System.out.println("Subdirectory: " + subDir.getName());
    }
}
```

4. **GetFiles**
   Para listar archivos, usa `listFiles()` con un filtro para verificar si es un archivo.

```
File dir = new File("C:\\MyDir");
File[] files = dir.listFiles(File::isFile);

if (files != null) {
    for (File file : files) {
        System.out.println("File: " + file.getName());
    }
}
```

5. **MoveTo**
   Para mover un directorio, usa `renameTo()`. Esto mueve el directorio completo, incluyendo sus contenidos.

```
File sourceDir = new File("C:\\MyDir\\SubDir");
File targetDir = new File("C:\\MyNewDir");

if (sourceDir.renameTo(targetDir)) {
    System.out.println("Directory moved to: " +
targetDir.getAbsolutePath());
} else {
    System.out.println("Failed to move directory");
}
```

3 Ficheros

En Java, estas operaciones se realizan principalmente con la clase `Files` de `java.nio.file`.

Crear un archivo:

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


public class FileOperations {
   public static void main(String[] args) {
      File file = new File("example.txt");
      try {
         if (file.createNewFile()) {
            System.out.println("File created: " + file.getName());
         } else {
            System.out.println("File already exists.");
         }
      } catch (IOException e) {
         System.out.println("An error occurred.");
         e.printStackTrace();
      }
   }
}
```

Copiar un archivo:

```
import java.nio.file.*;


public class FileOperations {
   public static void main(String[] args) {
      Path source = Paths.get("example.txt");
      Path target = Paths.get("copy_example.txt");

      try {
         Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
```

```
            System.out.println("File copied successfully.");
        } catch (IOException e) {
            System.out.println("An error occurred during copy.");
            e.printStackTrace();
        }
    }
}
```

Eliminar un archivo:

```
import java.io.File;

public class FileOperations {
    public static void main(String[] args) {
        File file = new File("example.txt");

        if (file.delete()) {
            System.out.println("File deleted: " + file.getName());
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

Mover un archivo:

```
import java.nio.file.*;

public class FileOperations {
    public static void main(String[] args) {
        Path source = Paths.get("example.txt");
        Path target = Paths.get("moved_example.txt");
```

```
        try {
            Files.move(source, target, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("File moved successfully.");
        } catch (IOException e) {
            System.out.println("An error occurred during move.");
            e.printStackTrace();
        }
    }
}
```

Abrir un archivo (Para lectura):

```
import java.io.File;
import java.util.Scanner;

public class FileOperations {
    public static void main(String[] args) {
        File file = new File("example.txt");

        try (Scanner reader = new Scanner(file)) {
            while (reader.hasNextLine()) {
                String line = reader.nextLine();
                System.out.println(line);
            }
        } catch (Exception e) {
            System.out.println("An error occurred while reading the file.");
            e.printStackTrace();
```

```
    }
  }
}
```

En Java, se utiliza principalmente la clase `File` para representar un archivo y obtener información sobre él. Ejemplos:

1. **Obtener el nombre y la ruta**:

```
File file = new File("example.txt");
System.out.println("File name: " + file.getName());
System.out.println("Full path: " + file.getAbsolutePath());
```

2. **Verificar si existe**:

```
File file = new File("example.txt");
System.out.println("Exists: " + file.exists());
```

3. **Tamaño del archivo**:

```
File file = new File("example.txt");
System.out.println("File size: " + file.length() + " bytes");
```

4. **Permisos del archivo**:

```
File file = new File("example.txt");
System.out.println("Readable: " + file.canRead());
System.out.println("Writable: " + file.canWrite());
System.out.println("Executable: " + file.canExecute());
```

5. **Última modificación**:

```
import java.util.Date;

File file = new File("example.txt");
System.out.println("Last modified: " + new
Date(file.lastModified()));
```

4 Concepto de flujo

En Java, el concepto de streams (flujos) es un mecanismo unificado que permite realizar operaciones de entrada y salida de datos (I/O), de manera secuencial, entre un origen de datos (data source) y un destino de datos (data sink). Este sistema es compatible con múltiples lenguajes de programación y sistemas operativos, y se utiliza en Java principalmente a través de las clases de los paquetes java.io y java.nio.

Los flujos en Java no están limitados a archivos; también se aplican a otras áreas como operaciones con buffers en memoria, sockets para comunicaciones en red, y más. Vamos a explorar su uso específicamente en el acceso a datos dentro de archivos, aunque su naturaleza es extrapolable a cualquier interacción de lectura o escritura secuencial.

Flujos en Java

Un stream es una secuencia de datos que se transmiten de forma secuencial desde un origen hasta un destino. En Java, existen dos tipos principales de flujos:

Flujos de Entrada (Input Streams):

Se utilizan para leer datos desde un origen, como un archivo, una red, o un buffer de memoria.

Ejemplo: Leer el contenido de un archivo de texto para procesarlo.

Flujos de Salida (Output Streams):

Se utilizan para escribir datos hacia un destino, como un archivo o una conexión de red.

Ejemplo: Escribir datos generados por una aplicación en un archivo.

Características de los Flujos

- Secuencialidad:

Los datos se procesan en el mismo orden en el que se transmiten.

  - En lectura, una vez que los datos se han leído, no es posible volver atrás para leerlos nuevamente sin reiniciar el flujo.
  - En escritura, los datos se escriben en el destino exactamente en el orden en que se generan.

- Estructura FIFO (First In, First Out):

El primer dato leído es el primero que se escribió, seguido por el segundo, y así sucesivamente.

- Transparencia:

El origen y el destino de los datos son abstractos para el programador. Esto significa que el mismo flujo puede usarse para interactuar con archivos, sockets, buffers en memoria, etc., sin cambiar el código base.

Tipos de Flujos en Java

- Flujos de Bytes:

Usados para datos binarios o sin procesar.

Entrada: InputStream y sus subclases como FileInputStream.

Salida: OutputStream y sus subclases como FileOutputStream.

Ejemplo de lectura de bytes:

```
import java.io.FileInputStream;
import java.io.IOException;

public class ByteStreamExample {
   public static void main(String[] args) {
      try (FileInputStream fis = new FileInputStream("example.txt")) {
         int byteData;
         while ((byteData = fis.read()) != -1) {
            System.out.print((char) byteData);
```

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

- Flujos de Caracteres:

Usados para datos textuales.

Entrada: Reader y sus subclases como FileReader.

Salida: Writer y sus subclases como FileWriter.

Ejemplo de lectura de caracteres:

```
import java.io.FileReader;
import java.io.IOException;

public class CharacterStreamExample {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("example.txt")) {
            int charData;
            while ((charData = reader.read()) != -1) {
                System.out.print((char) charData);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Flujos Bufferizados:

Mejoran el rendimiento al agrupar datos en bloques.

Clases: BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter.

Ejemplo con BufferedReader:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedStreamExample {
   public static void main(String[] args) {
      try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
         String line;
         while ((line = br.readLine()) != null) {
            System.out.println(line);
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Aplicaciones de los Flujos en Java

- Acceso a Archivos: Leer y escribir datos en archivos de texto o binarios.
- Comunicaciones en Red: Transmitir datos entre clientes y servidores mediante sockets.
- Manipulación de Buffers: Procesar datos almacenados temporalmente en memoria.

5 Flujos de entrada y salida de datos

En Java, la clase base para trabajar con flujos de bytes es InputStream (para entrada) y OutputStream (para salida). Estas clases se utilizan como punto de partida para flujos relacionados con archivos, comunicaciones en red, y otros medios. Para manejar archivos específicamente, Java proporciona la clase FileOutputStream (escritura) y FileInputStream (lectura), que permiten realizar operaciones de entrada y salida directamente sobre archivos.

Además, Java proporciona la clase File (de java.io) para representar y manipular archivos y directorios, pero la gestión de flujos se realiza mediante las clases mencionadas de InputStream y OutputStream. También se pueden usar modos específicos de apertura de archivos mediante sus constructores o combinando métodos de las clases de I/O.

Modos de Apertura de Archivos en Java

Aunque Java no utiliza directamente enumeraciones como FileMode en C#, las operaciones de apertura de archivos pueden controlarse utilizando combinaciones de clases de flujos y configuraciones específicas. Los modos comunes de apertura de archivos incluyen:

Modos de apertura equivalentes:

- Append

Abre el archivo si existe y posiciona el flujo al final del archivo para escribir nuevos datos. Si el archivo no existe, lo crea automáticamente.

Esto se realiza usando el constructor de FileOutputStream con el parámetro append en true.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class AppendExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt", true)) {
            String data = "Appending data to the file.\n";
            fos.write(data.getBytes());
```

```
            System.out.println("Data appended successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- CreateNew

Crea un archivo nuevo. Si el archivo ya existe, se lanza una excepción FileAlreadyExistsException. Esto se puede lograr utilizando la clase Files de java.nio.file.

```
import java.nio.file.*;

public class CreateNewExample {
    public static void main(String[] args) {
        Path path = Paths.get("newFile.txt");
        try {
            Files.createFile(path);
            System.out.println("File created successfully.");
        } catch (FileAlreadyExistsException e) {
            System.out.println("File already exists.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Truncate

Abre un archivo existente y lo vacía antes de escribir datos en él. Esto ocurre automáticamente al usar FileOutputStream sin el modo de append.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class TruncateExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt")) {
            String data = "This will overwrite existing content.\n";
            fos.write(data.getBytes());
            System.out.println("File truncated and written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Create

Crea un archivo nuevo o sobrescribe uno existente. Este comportamiento es similar al de Truncate, ya que FileOutputStream sobrescribe por defecto.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class CreateExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt")) {
            String data = "File created or overwritten.\n";
            fos.write(data.getBytes());
            System.out.println("File created or overwritten successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }
}
```

- OpenOrCreate

Abre un archivo si existe, o lo crea si no. Este es el comportamiento predeterminado de FileOutputStream.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class OpenOrCreateExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt", true)) {
            String data = "Data written to existing or new file.\n";
            fos.write(data.getBytes());
            System.out.println("File opened or created successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Tipos de Acceso a Archivos

En Java, el acceso a archivos se controla implícitamente mediante el tipo de flujo utilizado:

- Lectura (FileAccess.Read)

Usar FileInputStream para leer datos de un archivo.

```
import java.io.FileInputStream;
import java.io.IOException;

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

```
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int byteData;
            while ((byteData = fis.read()) != -1) {
                System.out.print((char) byteData);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Escritura (FileAccess.Write)

Usar FileOutputStream para escribir datos en un archivo.

```
import java.io.FileOutputStream;
import java.io.IOException;

public class WriteFileExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("example.txt")) {
            String data = "Writing data to file.";
            fos.write(data.getBytes());
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Lectura y Escritura (FileAccess.ReadWrite)

Para lectura y escritura simultánea, se puede usar RandomAccessFile o una combinación de flujos.

```
import java.io.RandomAccessFile;
import java.io.IOException;

public class ReadWriteExample {
    public static void main(String[] args) {
        try (RandomAccessFile raf = new RandomAccessFile("example.txt", "rw")) {
            // Escribir datos
            raf.write("This is read-write access.\n".getBytes());
            // Posicionar el puntero al inicio
            raf.seek(0);
            // Leer datos
            String line;
            while ((line = raf.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Veamos como leer y escribir un fichero byte a byte usando FileStream:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopyExample {

   public static void main(String[] args) {
      // Rutas del archivo de origen y destino
      String orig = "C:/Users/DAM/Documents/Sample.txt";
      String dest = "C:/Users/DAM/Documents/Copy.txt";

      // Usamos try-with-resources para asegurarnos de que los recursos se cierran automáticamente
      try (FileInputStream fileOrig = new FileInputStream(orig);
          FileOutputStream fileDest = new FileOutputStream(dest)) {

         // Leemos el archivo de origen byte por byte y lo escribimos en el archivo de destino
         int b;
         while ((b = fileOrig.read()) != -1) {
            fileDest.write(b);  // Escribimos el byte en el archivo de destino
         }

         System.out.println("Archivo copiado exitosamente.");

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

`BufferedWriter`, se usa para escribir caracteres en un archivo con una codificación determinada, como UTF-8. A continuación, te muestro cómo escribir un programa en Java para que el usuario introduzca frases por consola y se guarden en un archivo.

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

public class FileWriteExample {

   public static void main(String[] args) {
      // Ruta del archivo de salida
      String filePath = "C:/Users/DAM/Documents/Output.txt";

      // Crear el archivo de salida
      File file = new File(filePath);

      // Usamos Scanner para leer las frases del usuario
      Scanner scanner = new Scanner(System.in);

      // Usamos BufferedWriter para escribir en el archivo con codificación UTF-8
      try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileWriter(file, true), StandardCharsets.UTF_8))) {
         System.out.println("Introduce las frases (escribe 'salir' para terminar):");

         // Leemos las frases del usuario hasta que escriba "salir"
         String line;
         while (true) {
            line = scanner.nextLine();
            if ("salir".equalsIgnoreCase(line)) {
               break;
            }
            // Escribimos la línea en el archivo
            writer.write(line);
            writer.newLine(); // Escribimos un salto de línea después de cada frase
         }
         System.out.println("Frases guardadas exitosamente.");

      } catch (IOException e) {
         e.printStackTrace();
      } finally {
         scanner.close();
      }
   }
}
```

Para añadir texto a un archivo existente en Java, se puede utilizar la clase BufferedWriter junto con el constructor de FileWriter que acepta un parámetro true para habilitar el modo de añadir. Esto garantiza que el contenido se agregue al final del archivo sin sobrescribir el texto existente.

A continuación, te presento un ejemplo donde solicitamos al usuario que ingrese frases que se agregarán a un archivo existente:

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class AppendToFileExample {

   public static void main(String[] args) {
      // Ruta del archivo de salida
      String filePath = "C:/Users/DAM/Documents/Output.txt";

      // Usamos Scanner para leer las frases del usuario
      Scanner scanner = new Scanner(System.in);

      // Crear el archivo de salida si no existe
      File file = new File(filePath);

      // Usamos BufferedWriter para escribir en el archivo en modo append
      try (BufferedWriter writer = new BufferedWriter(new FileWriter(file, true))) {
         System.out.println("Introduce las frases para añadir al archivo (escribe 'salir' para terminar):");

         // Leemos las frases del usuario hasta que escriba "salir"
         String line;
         while (true) {
            line = scanner.nextLine();
            if ("salir".equalsIgnoreCase(line)) {
               break;
            }
            // Escribimos la línea en el archivo
            writer.write(line);
            writer.newLine(); // Añadimos un salto de línea
         }
         System.out.println("Frases añadidas exitosamente al archivo.");

      } catch (IOException e) {
         e.printStackTrace();
      } finally {
         scanner.close();
      }
   }
}
```

BufferedReader se utiliza para leer caracteres de un archivo de texto. En Java, puedes especificar la codificación mediante un InputStreamReader envolviendo un flujo de entrada (como un FileInputStream) y especificando la codificación, como UTF-8.

A continuación, se muestra cómo leer líneas de un archivo de texto utilizando BufferedReader en Java. La codificación utilizada en este caso es UTF-8.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;

public class FileReadExample {

    public static void main(String[] args) {
        // Ruta del archivo a leer
        String filePath = "C:/Users/DAM/Documents/Input.txt";

        // Usamos BufferedReader para leer el archivo
        File file = new File(filePath);

        // Usamos InputStreamReader para especificar la
codificación UTF-8
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(new                          FileReader(file),
StandardCharsets.UTF_8))) {
            String line;

            // Leemos el archivo línea por línea
            while ((line = reader.readLine()) != null) {
                System.out.println(line);  // Imprimimos cada
línea en la consola
            }

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

En Java, una manera común de leer un archivo completamente y obtener las líneas como elementos de un array es utilizando la clase Files junto con su método readAllLines. Este método lee todas las líneas de un archivo y las devuelve como una lista de cadenas (List<String>). Si necesitas un arreglo (array), puedes convertir esta lista a un array de cadenas utilizando el método toArray().

A continuación, te presento un ejemplo que muestra cómo leer completamente un archivo y obtener un array con las líneas del archivo:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

public class FileReadAllLinesExample {

    public static void main(String[] args) {
        // Ruta del archivo a leer
        String filePath = "C:/Users/DAM/Documents/Input.txt";

        try {
            // Leemos todas las líneas del archivo y las almacenamos en una lista
            List<String> lines = Files.readAllLines(Paths.get(filePath));

            // Convertimos la lista en un array
            String[] linesArray = lines.toArray(new String[0]);

            // Imprimimos las líneas leídas
            for (String line : linesArray) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

En Java, para leer todo el contenido de un archivo y almacenarlo en una sola cadena de caracteres (String), puedes utilizar la clase Files y su método readString(), que lee todo el contenido de un archivo y lo devuelve como una cadena. Si estás usando una versión anterior de Java que no soporte readString(), puedes utilizar un enfoque con BufferedReader o FileInputStream.

A continuación te muestro dos formas de hacerlo:

Método 1: Usando Files.readString() (Disponible desde Java 11)

Este método es muy sencillo y permite leer el contenido de un archivo completo y devolverlo como una sola cadena de texto.

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

public class FileReadToStringExample {

    public static void main(String[] args) {
        // Ruta del archivo a leer
        String filePath = "C:/Users/DAM/Documents/Input.txt";

        try {
            // Leemos todo el archivo y lo almacenamos en una sola
cadena de texto
            String content = Files.readString(Paths.get(filePath));

            // Imprimimos el contenido del archivo
            System.out.println(content);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Método 2: Usando BufferedReader (Para versiones anteriores de Java)

Si estás utilizando una versión de Java anterior a Java 11, puedes usar un BufferedReader para leer el archivo línea por línea y luego concatenar las líneas en una sola cadena.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.StringWriter;

public class FileReadToStringOldJavaExample {

    public static void main(String[] args) {
        // Ruta del archivo a leer
        String filePath = "C:/Users/DAM/Documents/Input.txt";

        StringBuilder content = new StringBuilder();

        try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String line;
            // Leemos el archivo línea por línea y agregamos cada
línea a la cadena
            while ((line = reader.readLine()) != null) {
                content.append(line).append(System.lineSeparator());
// Añadir la línea y salto de línea
            }

            // Imprimimos el contenido completo del archivo
            System.out.println(content.toString());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

6 Tareas comunes de E/S

Estas son las tareas comunes relacionadas con directorios y ficheros en Java, junto con las clases y métodos correspondientes que puedes usar para realizar esas operaciones. Java tiene clases específicas como File, Paths, Files, entre otras, para trabajar con directorios y archivos.

Tareas comunes con directorios

Crear un directorio	Método Directory.CreateDirectory Propiedad FileInfo.Directory
Crear un subdirectorio	Método DirectoryInfo.CreateSubdirectory
Cambiar el nombre de un directorio o moverlo	Método Directory.Move Método DirectoryInfo.MoveTo
Copiar un directorio	Cómo: Copiar directorios
Eliminar un directorio	Método Directory.Delete Método DirectoryInfo.Delete
Ver los archivos y los subdirectorios de un directorio	Cómo: Enumerar directorios y archivos
Determinar si existe un directorio	Método Directory.Exists

Tareas comunes sobre ficheros

Crear un archivo de texto	Método File.CreateText Método FileInfo.CreateText Método File.Create Método FileInfo.Create
Escribir en un archivo de texto	Cómo: Escribir texto en un archivo
Leer de un archivo de texto	Cómo: Leer texto de un archivo
Anexar texto a un archivo	Cómo: Abrir y anexar a un archivo de registro Método File.AppendText Método FileInfo.AppendText
Cambiar el nombre de un archivo o moverlo	Método File.Move Método FileInfo.MoveTo
Eliminar un archivo	Método File.Delete Método FileInfo.Delete
Copiar un archivo	Método File.Copy Método FileInfo.CopyTo
Obtener el tamaño de un archivo	Propiedad FileInfo.Length
Determinar si existe un archivo	Método File.Exists
Leer de un archivo binario	Cómo: Leer y escribir en un archivo de datos recién creado
Escribir en un archivo binario	Cómo: Leer y escribir en un archivo de datos recién creado
Recuperar la extensión de un nombre de archivo	Método Path.GetExtension
Recuperar la ruta de acceso completa de un archivo	Método Path.GetFullPath
Recuperar el nombre y la extensión de un archivo de una ruta de acceso	Método Path.GetFileName
Cambiar la extensión de un archivo	Método Path.ChangeExtension

7 Expresiones regulares. Patrones.

En Java, las expresiones regulares se manejan a través de la clase Pattern y Matcher del paquete java.util.regex. A continuación te explico cómo utilizar las expresiones regulares en Java, cómo se pueden aplicar las mismas funcionalidades mencionadas y cómo trabajar con las clases correspondientes.

¿Qué se puede hacer con expresiones regulares en Java?

1. Buscar patrones concretos de caracteres: Al igual que en C#, se puede utilizar regex para encontrar patrones específicos dentro de una cadena de texto.
2. Validar texto: Puedes validar si una cadena de texto cumple con un formato predefinido, como un correo electrónico o un número de teléfono.
3. Extraer, editar, reemplazar o eliminar subcadenas: Puedes buscar y reemplazar partes de texto o extraer información.
4. Generar informes: Utilizando las coincidencias encontradas, puedes agrupar y manipular información para generar un informe o hacer análisis de datos.

Sintaxis de Expresiones Regulares en Java

La sintaxis es muy similar a la de otras plataformas:

- \d: Cualquier dígito.
- \w: Cualquier letra, número o guion bajo.
- .: Cualquier carácter (excepto salto de línea).
- ^: Inicio de la cadena.
- $: Fin de la cadena.
- *: Cero o más repeticiones.
- +: Una o más repeticiones.
- []: Corchetes para definir un conjunto de caracteres (por ejemplo, [a-z]).
- (): Paréntesis para agrupar expresiones.

Funciones en Java para trabajar con expresiones regulares

En Java, se usan las clases Pattern y Matcher para trabajar con expresiones regulares.

1. Determinar si un texto sigue un patrón determinado

El método Pattern.matches() verifica si el texto sigue un patrón definido.

Ejemplo:

```
import java.util.regex.*;
public class ValidarCorreo {
   public static void main(String[] args) {
      String email = "usuario@dominio.com";
      boolean esValido = Pattern.matches("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$", email);
      System.out.println(esValido);  // Salida: true si el correo es válido
   }
}
```

2. Recuperar la primera coincidencia

Para obtener la primera coincidencia, usamos la clase Matcher con el método find(). El método matcher.group() devuelve la cadena que coincide con el patrón.

Ejemplo:

```
import java.util.regex.*;
public class PrimerCoincidencia {
   public static void main(String[] args) {
      String texto = "Mi teléfono es 123-456-7890.";
      Pattern patron = Pattern.compile("\\d{3}-\\d{3}-\\d{4}");
      Matcher matcher = patron.matcher(texto);

      if (matcher.find()) {
         System.out.println("Número de teléfono encontrado: " + matcher.group());
      } else {
         System.out.println("No se encontró un número de teléfono.");
      }
   }
}
```

3. Recuperar todas las coincidencias

Para obtener todas las coincidencias, utilizamos Matcher.find() dentro de un bucle.

Ejemplo:

```
import java.util.regex.*;
```

```
public class TodasCoincidencias {
   public static void main(String[] args) {
      String texto = "Mis números son 123-456-7890 y 987-654-3210.";
      Pattern patron = Pattern.compile("\\d{3}-\\d{3}-\\d{4}");
      Matcher matcher = patron.matcher(texto);

      while (matcher.find()) {
         System.out.println("Número encontrado: " + matcher.group());
      }
   }
}
```

4. Reemplazar el texto que coincide con un patrón

Para reemplazar una coincidencia con otro texto, usamos el método replaceAll() de la clase Matcher.

Ejemplo:

```
import java.util.regex.*;
public class ReemplazarTexto {
   public static void main(String[] args) {
      String texto = "Mi teléfono es 123-456-7890 y mi código postal es 987-654-3210.";
      String textoModificado = texto.replaceAll("\\d{3}-\\d{3}-\\d{4}", "XXX-XXX-XXXX");
      System.out.println(textoModificado);
   }
}
```

Resumen de métodos clave en Java

- Pattern.matches(regex, texto): Verifica si un texto completo coincide con una expresión regular.
- Pattern.compile(regex): Compila una expresión regular en un patrón, que se puede usar para crear un objeto Matcher.
- matcher.find(): Busca la siguiente coincidencia en el texto.

- matcher.group(): Devuelve la subcadena que coincide con el patrón.
- texto.replaceAll(regex, replacement): Reemplaza todas las coincidencias del patrón en el texto con una cadena de reemplazo.

En Java, las expresiones regulares son herramientas poderosas que nos permiten realizar búsquedas, validaciones y manipulaciones complejas de cadenas de texto. A continuación, te explico cómo funcionan las reglas generales para construir patrones en expresiones regulares y cómo puedes aplicarlas en Java.

Reglas Generales para Construir Patrones

Las reglas que describen cómo crear patrones son muy similares en Java y otros lenguajes que soportan expresiones regulares. A continuación, explico algunos de los patrones más comunes y cómo se utilizan en Java:

1. Coincidencias Básicas

- a: Coincide exactamente con el carácter a.
- [abc]: Coincide con cualquiera de los caracteres a, b o c.
- [a-z]: Coincide con cualquier letra minúscula entre a y z.
- [A-Za-z]: Coincide con cualquier letra mayúscula o minúscula entre A-Z o a-z.
- [0-9]: Coincide con cualquier dígito entre 0 y 9.

2. Cuantificadores

- A?: Coincide con el carácter A, una vez o ninguna vez.
- A*: Coincide con el carácter A, cero o más veces.
- A+: Coincide con el carácter A, una o más veces.
- A{2}: Coincide con exactamente dos veces el carácter A.
- A{3,}: Coincide con tres o más veces el carácter A.
- A{2,4}: Coincide con entre 2 y 4 veces el carácter A.

3. Combinaciones

- [a-z]{1,4}[0-9]+: Coincide con entre 1 y 4 letras minúsculas, seguido de uno o más dígitos.

4. Negación

- [^abc]: Coincide con cualquier carácter que no sea a, b o c.
- .: Coincide con cualquier carácter, excepto el salto de línea.

5. Clases de Caracteres

- \d: Coincide con cualquier dígito (equivalente a [0-9]).
- \D: Coincide con cualquier carácter que no sea un dígito (equivalente a [^0-9]).
- \s: Coincide con un espacio en blanco (incluye espacios, tabulaciones, saltos de línea, etc.).
- \S: Coincide con cualquier carácter que no sea un espacio en blanco.
- \w: Coincide con cualquier carácter de palabra (letras, dígitos, y guion bajo: [a-zA-Z_0-9]).
- \W: Coincide con cualquier carácter que no sea de una palabra (cualquier cosa que no sea alfanumérica o guion bajo).

6. Operadores Lógicos (OR)

- X|Y: Coincide con X o Y. Por ejemplo, a|b coincidirá con a o b.

7. Límites

- ^: Marca el comienzo de la cadena.
- $: Marca el final de la cadena.
- \b: Coincide con un límite de palabra (el borde de una palabra).
- \B: Coincide con cualquier lugar que no sea un límite de palabra.

Ejemplos Prácticos

1. Buscar una palabra específica

Si quieres buscar la palabra "data" como parte de una palabra o una palabra completa, puedes usar las siguientes expresiones regulares:

- Buscar "data" en cualquier parte de una palabra: data.
- Buscar "data" como una palabra individual: \bdata\b.

2. Validación de un código postal

Para verificar si un texto tiene el formato de un código postal (por ejemplo, 5 dígitos consecutivos), puedes usar el siguiente patrón:

- Código postal de 5 dígitos: ^\d{5}$.

3. Validar una dirección de correo electrónico

El siguiente patrón de expresión regular valida direcciones de correo electrónico simples:

- Correo electrónico: ^[^@\s]+@[^@\s]+\.[^@\s]+$.

Implementación de Expresiones Regulares en Java

En Java, se utilizan las clases Pattern y Matcher para trabajar con expresiones regulares.

1. Validar un correo electrónico con una expresión regular

```
import java.util.regex.*;
public class ValidarEmail {
   public static void main(String[] args) {
      String email = "usuario@dominio.com";
      String regex = "^[^@\\s]+@[^@\\s]+\\.[^@\\s]+$";  // Expresión regular para correo electrónico
      Pattern pattern = Pattern.compile(regex);
      Matcher matcher = pattern.matcher(email);
      if (matcher.matches()) {
         System.out.println("Correo válido");
      } else {
         System.out.println("Correo no válido");
      }
   }
}
```

2. Buscar un código postal

```
import java.util.regex.*;
public class ValidarCodigoPostal {
   public static void main(String[] args) {
      String texto = "Mi código postal es 12345.";
      String regex = "^\\d{5}$";  // Expresión regular para código postal

      Pattern pattern = Pattern.compile(regex);
      Matcher matcher = pattern.matcher(texto);
      if (matcher.find()) {
         System.out.println("Código postal válido: " + matcher.group());
      } else {
         System.out.println("Código postal no válido");
      }
```

```
    }
}
```

3. Buscar una palabra específica en un texto

```
import java.util.regex.*;
public class BuscarPalabra {
    public static void main(String[] args) {
        String texto = "Este es un ejemplo con la palabra data en él.";
        String regex = "\\bdata\\b";  // Expresión regular para buscar la palabra "data"
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(texto);
        if (matcher.find()) {
            System.out.println("Palabra encontrada: " + matcher.group());
        } else {
            System.out.println("Palabra no encontrada");
        }
    }
}
```

7.1 Editar cadenas

En Java, al igual que en otros lenguajes, las expresiones regulares permiten realizar sustituciones de texto usando el método replaceAll. Este método es útil para editar cadenas de texto, por ejemplo, para transformar fechas o para reemplazar patrones en un texto.

Ejemplo: Convertir fechas de formato español (dd/MM/yyyy) a formato canónico (yyyy-MM-dd)

Vamos a suponer que tenemos una cadena de texto con fechas en formato dd/MM/yyyy (por ejemplo, "25/12/2024") y queremos convertirlas al formato yyyy-MM-dd (por ejemplo, "2024-12-25"). Usaremos una expresión regular para capturar los componentes de la fecha y luego los reorganizaremos en el formato deseado.

1. Expresión Regular

La expresión regular que utilizaremos para capturar la fecha en formato dd/MM/yyyy será:

- (\d{2}): Captura el día (dos dígitos).
- (\d{2}): Captura el mes (dos dígitos).
- (\d{4}): Captura el año (cuatro dígitos).

Por lo tanto, la expresión regular completa será: (\d{2})/(\d{2})/(\d{4}).

2. Método replaceAll

El método replaceAll de la clase String permite reemplazar una subcadena (que coincide con una expresión regular) por otra subcadena. Usaremos la expresión regular para capturar los tres componentes de la fecha y luego los organizamos en el nuevo formato.

```
import java.util.regex.*;

public class ConvertirFecha {

   public static void main(String[] args) {

      // Fecha en formato dd/MM/yyyy

      String fechaOriginal = "25/12/2024";

      // Expresión regular para capturar dd, MM, yyyy

      String regex = "(\\d{2})/(\\d{2})/(\\d{4})";

      // Reemplazamos la fecha al formato yyyy-MM-dd

      String fechaConvertida = fechaOriginal.replaceAll(regex, "$3-$2-$1");

      // Mostramos el resultado

      System.out.println("Fecha original: " + fechaOriginal);

      System.out.println("Fecha convertida: " + fechaConvertida);

   }

}
```