Contents

4.1 What is Java NIO (Non-blocking Input/Output) API

- Buffer-oriented.
- It deals with data in blocks.
- Non-blocking IO operation.

The "**java.nio**" package defines interfaces and classes for NIO operating with files and file systems.

4.2 NIO Path & Files

- **Path** interface – defines location of a file or directory.
- **Files** class – provides several static methods for manipulating files or directories and those methods mostly works with Path object.

Constructing a Path

- Path represents an object that locates a file in the file system.
- To Create Path object
  - Paths.get() method is used in Java 8.
  - Path.of() method is used in Java 11.

Example

```
// Java < 11
Path path = Paths.get("E:\\ProgrammingFundamentals\\Files\\readme.txt");
System.out.println(path);
// Java11+
path = Path.of("E:\\ProgrammingFundamentals\\Files\\readme.txt");
System.out.println(path);
path = Path.of("E/programmingfundamentals/files/readme.txt");
System.out.println(path);
path = Path.of("E","programmingfundamentals","files","readme.txt");
System.out.println(path);
path = Path.of("E", "programmingfundamentals","files").resolve("readme.txt");
System.out.println(path);
path = Path.of("./src/pkg1/PathApi.java"); //relative or current path
System.out.println(path.toAbsolutePath()); //absolute path
path = Path.of("src", "pkg1","PathApi.java");
System.out.println(path);
```

Basic File Operations

- Files class contains static methods that work on files and directories.
- It works with Path instances.

1. Creating File or Directory

- Files.createFile() – It creates a new and empty file, failing if the file already exists.

Example

```
// create file
Path path = Path.of("src/readme.txt");
if(Files.exists(path))
        System.out.println("File already exists");
else
{
        Files.createFile(path);
        System.out.println("File created");
}
// create directory
path = Path.of("./test"); // current working dir
        if(Files.notExists(path))
                Files.createDirectories(path);
```

2. Deleting a File or Directory

- deleteIfExists() – It deletes the file if it exists.
- delete() – It deletes the file or throws an exception if deletion fails.

Example – 1

```
Path path = Path.of("src/readme.txt");
boolean fileDeleted = Files.deleteIfExists(path);
if(fileDeleted)
        System.out.println("File deleted");
else
        System.out.println("File does not exist");
```

Example – 2

```
// delete empty directory
Path path = Path.of("E:\\ProgrammingFundamentals\\Files");
boolean result = Files.deleteIfExists(path);
System.out.println((result) ? "Deleted" : "Failed");
// delete non-empty directory
path = Path.of("e:/programmingfundamentals/test");
if(Files.exists(path)) {
        Files.walk(path)  //list all the files
                .map(p -> p.toFile()) // convert Path to File
                .forEach(f -> f.delete()); // delete File

        Files.delete(path); // delete dir
}
```

3. Copying File or Directory

- copy(Path, Path, CopyOption...) – The copy fails if the target file exists, unless the REPLACE_EXISTING option is specified.

Example

```
Path source = Path.of("src/readme.txt");
Path dest = Path.of("src/another.txt");

Files.copy(source, dest); // if another.txt not exist, automatically create new file

source = Path.of("src/myfile.txt");

Files.copy(source, dest,StandardCopyOption.REPLACE_EXISTING);
```

4. Moving File or Directory

- move(Path, Path, CopyOption...) method – The move fails if the target file exists, unless the REPLACE_EXISTING option is specified.

Example

```
Path source = Path.of("src/myfile.txt");
Path dest = Path.of("src/pkg1/myfile.txt");

Files.move(source, dest, StandardCopyOption.REPLACE_EXISTING);
```

5. Listing Contents of Directory

- Files.list() – non-recursive (it does not traverse the subdirectories)
- Files.walk() – recursive (it can traverse the subdirectories)

Both method returns a stream of Path object.

Example – Files.list()

```
Path path = Path.of(".");
try(Stream<Path> fileList = Files.list(path)){
        fileList.forEach(System.out::println);
}catch (Exception e) {
        // TODO: handle exception
}
```

Example – Files.walk()

```
Path path = Path.of(".");
try(Stream<Path> fileList = Files.walk(path)){
        System.out.println("*** All files and directories ***");
        fileList.forEach(System.out::println);

        System.out.println("*** Only directories ***");
        Files.walk(path)
                .filter(Files::isDirectory)
                .forEach(System.out::println);

        System.out.println("*** only custom files ***");
        Files.walk(path)
                .filter(p -> p.toString().contains(".java"))
                .forEach(System.out::println);
}catch (Exception e) {
        // TODO: handle exception
}
```

4.3 Reading File

- Commonly used methods to read file –
    1. Files.lines() – return a Stream (Java 8)
    2. Files.readString () – returns a String (Java 11), max file size 2G.
    3. Files.readAllBytes () – returns a byte [] (Java 7), max file size 2G.
    4. Files.readAllLines () – returns a List<String> (Java 8)
    5. BufferedReader, a classic old friend (Java 1.1 -> forever)
    6. FileChannel – performs faster than standard I/O when a large file is read.

Files.lines

- In the method, the Stream has the reference to the open file.
- The file will only be closed when the stream is closed.
- Using the Files.lines in a try-with-resource statement is a common way.
- By default, it uses UTF-8 character encoding.

Example

```
Path path = Path.of("src/employee.txt");

try(Stream<String> data = Files.lines(path)){ // default charset is UTF_8.
        List<String> list = data.filter(d -> d.contains("Aung"))
                                .collect(Collectors.toList());
        list.forEach(a -> System.out.println(a));

}catch (Exception e) {

}
// with charset
try(Stream<String> data = Files.lines(path,StandardCharsets.UTF_8)){
        data.map(d -> d.toUpperCase()).forEach(System.out::println);
}catch (Exception e) {

}
```

- For reading in a small text file, Stream can be easily converted into List<String>.
- This conversion will throw java.lang.OutOfMemoryError if the Stream size is larger than the running JVM heap size (E.g. reading a large file).

Example

```
// using parallel stream
Stream<String> data = Files.lines(path).parallel();
List<String> list = data.collect(Collectors.toList());

// using BufferedReader
try(BufferedReader reader = Files.newBufferedReader(path)){
        String currentLine = null;
        while((currentLine = reader.readLine()) != null) {
                // do your work
        }
}catch(Exception e) {

}
```

Files.readAllLines

- It return List<String> and its default character encoding is UTF8.
- If the file size is larger than the running JVM heap size, it will throw java.lang.OutOfMemoryError.
- This method ensures that the file is closed when all contents have been read or an I/O error.

Example

```
class Employee{
        private int id;
        private String name;
        private String address;
        // getter/setter
        public static Employee getEmployeeFromLine(String line) {
                String[] tmp = line.split("\t");
                Employee emp = new Employee();
                emp.setId(Integer.parseInt(tmp[0]));
                emp.setName(tmp[1]);
                emp.setAddress(tmp[2]);
                return emp;
        }
}
List<String> lines = Files.readAllLines(Path.of("employee.txt"));
List<Employee> empList = lines.stream()
                        .map(Employee::getEmployeeFromLine)
                        .collect(Collectors.toList());
empList.forEach(System.out::println);
```

// employee.txt

1	Aung Aung	Yangon
2	Thae Thae Aung	Monywa
3	Yee Wai Khaing	Mandalay
4	Barani Aung	Mandalay

// Output

```
Employee [id=1, name=Aung Aung, address=Yangon]
Employee [id=2, name=Thae Thae Aung, address=Monywa]
Employee [id=3, name=Yee Wai Khaing, address=Mandalay]
Employee [id=4, name=Barani Aung , address=Mandalay]
```

Files.readAllBytes ()

- It is similar to Files.readAllLines() method but it returns array of byte [].

Example

```
byte[] bytes = Files.readAllBytes(filePath);
System.out.println(new String(bytes));
```

Files.readString()

- This method reads all content from a file into a string.
- It ensures that the file is closed when all content has been read.
- Default character encoding is UTF-8
- If the reading file size exceeds JVM's memory size, it will throws java.lang.OutOfMemoryError.

Example

```
Path path = Path.of("employee.txt");
String data = Files.readString(path);
System.out.println(data);
```

Reader with BufferReader

- It works well in reading small and large files.

Example

```
Path path = Path.of("employee.txt");

// read line by line in Java 8
try(BufferedReader br = Files.newBufferedReader(path)){
        List<String> list = br.lines()
                                .filter(d -> d.contains("Aung"))
                                .collect(Collectors.toList());
        System.out.println(list);
}catch (Exception e) { }

// read line by line before java 8
try(BufferedReader br = Files.newBufferedReader(path)){
        String line = null;
        List<String> list = new ArrayList<>();
        while((line = br.readLine()) != null) {
                if(line.contains("Aung"))
                        list.add(line);
                }
        }catch (Exception e) { }
```

4.4 Writing File

There are multiple ways of writing files in Java. Some of these are –

1. **Files.write**
   - It can be used to write both bytes and characters.
   - It can write a List of Strings at a single statement.
   - Default character encoding is UTF-8.
   - It ensures that the file is closed when all the bytes have been written.
   - The file will automatically be created (and truncated if it already exists).

Example

```
List<String> cities = List.of("Yangon","Mandalay","Pyin Oo Lwin","Insein","Hledan","Hlaing");
Path path = Path.of("city_data1.txt"); // if not exist, auto create. if exist, override it

Files.write(path, cities);
Files.write(path, "Monywa".getBytes(),StandardOpenOption.APPEND);
```

2. **Files.writeString**
   - It was introduced in Java 11.
   - It is used to directly write String contents into a file.
   - Default charset is UTF-8.
   - It makes auto-close file resources.

Example

```
Path path = Path.of("student.txt");
String data = """
                    Name: Aung Aung
                    Address: Yangon
                    Age: 26
            """;
Files.writeString(path, data,StandardOpenOption.CREATE_NEW);
```

   - Both Files.write () and Files.writeString () can be set up Open Options.
   - By default, the file will automatically be created (and truncated if it already exists).

Standard Open Option Modes

CREATE	It will create a new file if the file does not already exist. If already exist, existing data will be overridden.
CREATE_NEW	It will create a new file and write the data. It will fail if the file already exists.
WRITE	It will open a file for writing. If the file does not exist, it will throw NoSuchFileException.
APPEND	It will write the data to the end of the file.
TRUNCATE_EXISTING	It will make truncating or clearing out all the existing data.

- Both Files.write () and Files.writeString() are suitable for only small files.
- For large files, BufferWriter and FileChannel should be used.

3. BufferWriter

- Its IO process is similar to BufferReader.
- It supports buffering capabilities.

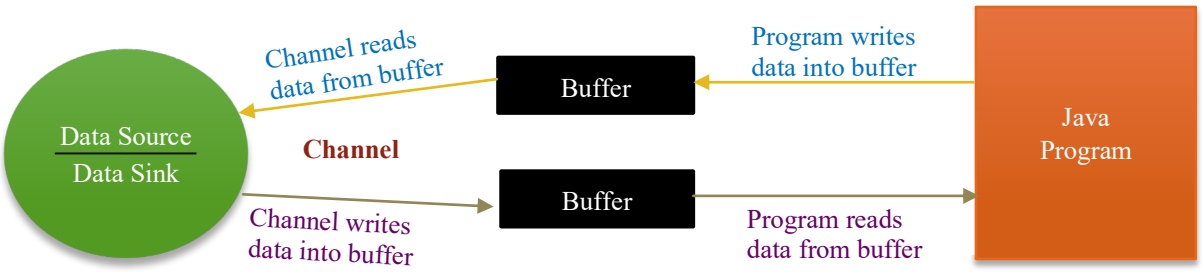
Example

```
Path path = Path.of("test/tester1.txt");
try(BufferedWriter writer = Files.newBufferedWriter(path)){
        writer.write("BufferedWriter is the simplest way of writing textual data to a File.");
        writer.write("\nIt buffers characters to improve performance.");
        writer.newLine();
        writer.write("As it buffers before writing, ");
        writer.flush();
        writer.write("so it results in less IO operations.");
}catch (Exception e) {

}
```

4.5 What is FileChannel

- FileChannel can be faster than standard IO when the large files are processed.
- Always runs in blocking mode.
- Thread safe, as read/write can be done asynchronously.



How to read data with FileChannel

Some steps are needed to read data from a file and to write data to a file using NIO.

1. Get a FileChannel object.
2. Create a ByteBuffer object to read data from the file.
3. Call the read () method of the FileChannel object by passing a ByteBuffer object.
4. Call the flip () method to read data from the buffer.
5. Read data from the ByteBuffer into your program.
6. Close the channel using its close () method.

Example – Read Small File with ByteBuffer and FileChannel

```
try(RandomAccessFile inputFile = new RandomAccessFile("employee.txt", "r");
        FileChannel channel = inputFile.getChannel()){
        long fileSize = channel.size();
        // create buffer
        ByteBuffer buffer = ByteBuffer.allocate((int)fileSize);
        // read data from channel to buffer
        channel.read(buffer);
        // prepare to read data from buffer
        buffer.flip();
        String data = new String(buffer.array(),0,buffer.limit());
        System.out.println(data);
}
```

Example – Read Large File with ByteBuffer and FileChannel

```
try(FileChannel channel = FileChannel.open(Path.of("employee.txt"),
StandardOpenOption.READ)){
        int bufferSize = 1024;
        if(bufferSize > channel.size())
                bufferSize = (int) channel.size();

        ByteBuffer buffer = ByteBuffer.allocate(bufferSize);

        while(channel.read(buffer) > 0) { // if reach end-of-file, return -1
            buffer.flip(); // Flip the buffer before we can read data from it
            String data = new String(buffer.array(),0,buffer.limit());
            System.out.println("Content: " + data);
            buffer.clear();
        }
}catch (Exception e) {
        // TODO: handle exception
}
```

How to write data with FileChannel

To write data to a file using Channel, the following steps are needed.

1. Obtain a FileChannel via an InputStream, OutputStream, or a RandomAccessFile. It can be also obtained by using FileChannel.open() method.
2. Create a ByteBuffer and then fill it with data.
3. Call the flip () method to read data from buffer and pass it as an argument of the write () method of the FileChannel.
4. After writing process is done, the resource is needed to close.

Example - Write Small data using ByteBuffer and RandomAccessFie

```
String data = """
                FileChannel can be faster than
                Standard I/O if you're dealing with large files.
        """;

try(RandomAccessFile inputFile = new RandomAccessFile("data.txt", "rw");
        FileChannel fileChannel = inputFile.getChannel()){
        //create byte buffer with sufficient capacity;
        byte[] byteData = data.getBytes();
        ByteBuffer buffer = ByteBuffer.allocate(byteData.length);
        // transfer data to the buffer
        buffer.put(byteData);
        // read data from buffer to prepare for channel write
        buffer.flip();
        // write sequence of byte to the channel from the buffer
```

```
            fileChannel.write(buffer);
}catch (Exception e) {
            // TODO: handle exception
}
```

Example – Write large data

```
Path path = Path.of("data.txt");
try(RandomAccessFile file = new RandomAccessFile(path.toFile(), "rw")){
            String largeData = "Many data .....";
            // Get file channel
            FileChannel channel = file.getChannel();
             // Get direct byte buffer access
MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_WRITE, 0, 4096 * 8);
            //Write the content using put methods
           buffer.put(largeData.getBytes());

}catch (Exception e) {

}
```

Further Reading

1. https://www.javatpoint.com/java-nio
2. https://www.happycoders.eu/java/how-to-write-files-quickly-and-easily/
3. https://docs.oracle.com/javase/9/docs/api/java/nio/file/Files.html
4. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/file/Files.html
5. https://docs.oracle.com/javase/tutorial/essential/io/file.html
6. https://34codefactory.medium.com/java-how-to-write-to-a-file-code-factory-339635bfb765
7. https://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileChannel.html

Assignment 1

s

THANK YOU