Java File IO

Lab setup

- Create a new Java project called "JavaFileIO"
- Add a main class to the project.
- In the root directory of the project, create a directory called “data”.
- In the “data” directory, add some arbitrary files.

I/O classes

Most classes you will need to handle I/O in Java are part of the `java.io` and `java.nio` packages.

File

The `File` class is a representation of a file in a filesystem. This class is in the `java.io` package. A `File` object can be created by passing a path name as a `String` to the constructor. Different systems use different separators for paths. Java provides a portable way to write path names with static fields `separator` and `pathSeparator`. The `separator` should be used when writing path names. The `pathSeparator` is used when giving multiple paths.

The default directory while we are working in Eclipse is the project directory.

1. Create a File object called `myFile` given the path "`./data/quotes.txt`".

```
File myFile = new File("." + File.separator + "data" + File.separator + "quotes.txt");
```

Figure 1 Creating a File object.

2. You may have noticed this file does not already exist. We can’t operate on a file that does not exist. We can check that it exists using the method `exists()`. The method `createNewFile()` also checks for existence before attempting to create the file.

```
boolean fileCreated;
try {
    fileCreated = myFile.createNewFile();
} catch (Exception ex) {
    ex.printStackTrace();
}
```

Figure 2 Create a file with a File object.

3. Directories are just files to a filesystem, so we can use `File` to operate on a directory as well. Create a new `File` object for the `data` directory.

```
File dataDir = new File("." + File.separator + "data");
```

Figure 3 File object for directory ./data

4. We can list the contents of the directory using `list()` or `listFiles()`. The `list()` method will return an array of Strings and `listFiles()` will return an array of `File` objects.

```
for (File f : dataDir.listFiles()) {
    System.out.println(f.getAbsolutePath());
}
```

Figure 4 Listing the files is a given directory.

5. A `Path` object (discussed next) can be obtained from a `File` using the `toPath()` method.

```
Path dataPath = dataDir.toPath();
```

Figure 5 Obtaining a Path object from a File object.

Path

The `Path` interface represents a system independent file path. It is part of the `java.nio` package. To get much of the same functionality as `File` we also need the `java.nio.Files` static class, `java.nio.file.FileSystem`, and `java.nio.file.FileSystems`.

6. Create a `Path` object representing the `data` directory.

```
Path data = FileSystems.getDefault().getPath(".","data");
```

Figure 6 Creating a Path object

The `getPath()` static method returns a `Path` when joining the provided Strings into a path string e.g., "./data" in this case.

7. List the entries in the `data` directory.

```
try (DirectoryStream<Path> stream = Files.newDirectoryStream(data)){
    for (Path entry : stream) {
        System.out.println(entry.getFileName());
    }
} catch (IOException e){
    e.printStackTrace();
}
```

Figure 7 Listing the files in a given directory

8. A `File` object can be obtained from a `Path` using the `toFile()` method.

```
File fromPath = data.toFile();
```

Figure 8 Obtaining a File object from a Path object.

Writing to files

Using `File` and `FileWriter` class

The `FileWriter` class writes text to character files. It can be instantiated from a `File` object or directly from a `String` of a file name. This file does not have to already exist in the file system.

```
FileWriter writer = new FileWriter(new File("." + File.separator + "myText.txt"));
```

Figure 9 Creating a FileWriter object

We use the `write(String)` method to write to the file.

```
writer.write("I told Java to write this line\n")
```

Figure 10 Writing a string to a file.

Then we must close the `FileWriter`

```
writer.close();
```

Figure 11 Closing a FileWriter object

We may also use the try-with-resources block because we need to handle possible exceptions anyway and it will handle closing the `FileWriter`.

```
try (FileWriter fw = new FileWriter(new File("." + File.separator + "myText.txt"))){
    fw.write("I wrote this inside a try-with-resources");
}
catch (IOException ex){
    ex.printStackTrace();
}
```

Figure 12 Create FileWriter object in try-with-resources block.

If we expect to make many small writes to a file it may be more efficient to use the `BufferedWriter` class. Figure 13 shows creating a `BufferedWriter` object using the `FileWriter` object from Figure 9. `BufferedWriter` is used like `FileWriter`.

```
BufferedWriter bw = new BufferedWriter (writer);
```

Figure 13 Creating a BufferedWriter object

Graded Submission Task 1

We will write some text from user input into a file called "userEcho.txt".

1. Create a `FileWriter` object for "userEcho.txt"
2. Write a simple loop to take multiple lines of text (delimited by the newline character) from the console and write them to the file. Do this for 10 lines of text. You can see the output format in Figure 13

```
<first_line>
<second_line>
...
<tenth_line>
```

Figure 14 Output format

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 1. You will submit this copy at the end of this lab. Reading from files

Reading from files

Java provides `FileReader` and `BufferedReader` that read from character streams. These are similar to the `FileWriter` and `BufferedWriter` classes discussed in the previous section. These classes can only read lines or portions of the stream. You will still need to parse the Strings to get non-character/string values. Luckily, we can create a `Scanner` object to do the parsing for us. If reading from the file is potentially costly, we can create a `BufferedReader` to pass to the Scanner.

Scanner

The `Scanner` class can take either a `Path` or `File` object in the constructor then we can take advantage of the parsing it does for us as we would if we were reading from `System.in`.

Graded Submission Task 2

1. Create a `Scanner` object to read from the "userEcho.txt" file from Task 1
2. Read each line from "userEcho.txt" and output each line in all upper-case

3. Use a `FileWriter` to write some numbers to a file, "userNumbers.txt" e.g. 0 1.4 566 10
4. Create a `Scanner` object to read from "userNumbers.txt"
5. Read each number from the file and output its value divided by 2

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 2. You will submit this copy at the end of this lab.

More methods from Files class using Path API

- `Files.newBufferedReader(Path)`: Obtain a BufferedReader from a given Path.
- `Files.newBufferedWriter(Path)`: Obtain a BufferedWriter from a given Path.
- `Files.newDirectoryStream(Path)`: Obtain a DirectoryStream<Path> from a given Path.
- `Files.createFile(Path, FileAttribute<?>… attrs)`: Create a file at the given Path.
- `Files.createTempFile()`: Create a temporary file in the default temp directory.
- `Files.createTempDirectory(String, FileAttribute<?>… attrs)`: Create a new directory in temp directory using the String for as a prefix to generate the name.
- `Files.exists(Path)`: Tests whether a file exists.
- `Files.readAllLines(Path)`: Read all lines from a file into a List<String>
- `Files.write(Path, Iterable<? extends CharSequence>, OpenOption… options)`: Write lines of test to a file.

Lab wrap-up

In this lab we discussed reading and writing to files using Java. We have two options for representing files in the file system using the Path or File classes. For writing to files we can objects of either to the `FileWriter` class or `BufferedWriter` class. A `FileWriter` is instantiated with a `File` object. If we expect to make many small writes to the file we can create a `BufferedWriter` from the `FileWriter`.

We can use the `Scanner` class with a `File` or `Path` object to read from files while taking advantage of the parsing that `Scanner` does. If reading from the file is potentially costly, we can create the `Scanner` with a `BufferedReader` as well.

Submission notes for graded tasks:

- Submit a zipped folder containing the Eclipse projects for each graded task.
- For each project, ensure you have included the src folder containing the .java file, the .classpath file, and the .project file.
- Ensure that each project is named appropriately to correspond with each task.
- Zip the projects together and name the zipped folder "Lab21_firstName_lastName", where cslogin is your login ID for the computers at the Department of Computer Science at ODU.
- Submit the zipped file to the respective Canvas link.

Rubrics	
Task 1 (points allotted) - Compilation (2) - Create a FileWriter object (2) - Reading lines from the console (2) - Output file echoes console input (4)	Task 2 (points allotted) - Compilation (2) - Create Scanner object for "userEcho.txt" (2) - Output lines from "userEcho.txt" in all upper-case (2) - Create Scanner object for "userNumbers.txt" (2) - Output numbers from "userNumbers.txt" divided by 2 (2)

Appendix

This appendix will include some useful snippets of code patterns.

Reading from a FileReader/BufferedReader until end-of-file

```
// Assume br is a BufferedReader
String line;
while ( (line = br.readLine()) != null) {
    // Do something with line
    System.out.println(line);
}
```

Reading files from a directory, skipping any failures

```
try (DirectoryStream<Path> dir = Files.newDirectoryStream(Path.of("data"))) {
    for (Path p : dir) {
        try (BufferedReader br = Files.newBufferedReader(p)) {
            br.readLine();
        } catch (IOException e) {
            System.err.println("Error reading file %s".formatted(p.toString()
));
            System.err.println("Skipping...");
        }
    }
} catch (IOException e) {
    System.err.println("Terminate reading directory");
}
```