Module 1-18

File Output

Objectives

- Students should be able to investigate File and Directory metadata using the File class
- Students should be able to write text data to a file
- Students should be able to explain the concept of buffering in File I/O
- Students should understand the importance of releasing external resources

File Object

- Java’s representation of file or directory path name
  - File and directory names have different formats depending on OS
- Has methods for working with:
  - Path name
  - Deleting and renaming files
  - Creating new directories
  - Listing contents of directory
  - Listing common attributes of files and directories

File Object

```
import java.io.File;  // Import the File class

public class GetFileInfo {
  public static void main(String[] args) {
    File myFile = new File("filename.txt");
    if (myFile.exists()) {
      System.out.println("File name: " + myFile.getName());
      System.out.println("Absolute path: " + myFile.getAbsolutePath());
      System.out.println("Writeable: " + myFile.canWrite());
      System.out.println("Readable " + myFile.canRead());
      System.out.println("File size in bytes " + myFile.length());
    } else {
      System.out.println("The file does not exist.");
    }
```

Java Output

Java has the ability to communicate data back to the user. Consider some of these methods:

- Using System.out.println() that sends a message to the console.
- Write data to a database (Module 2).
- Transmit data to an API (Module 3).
- Send a HTML view back to the user (Module 4).

For today, we will focus on something simpler, writing data back to a text file.

File class: create a directory.

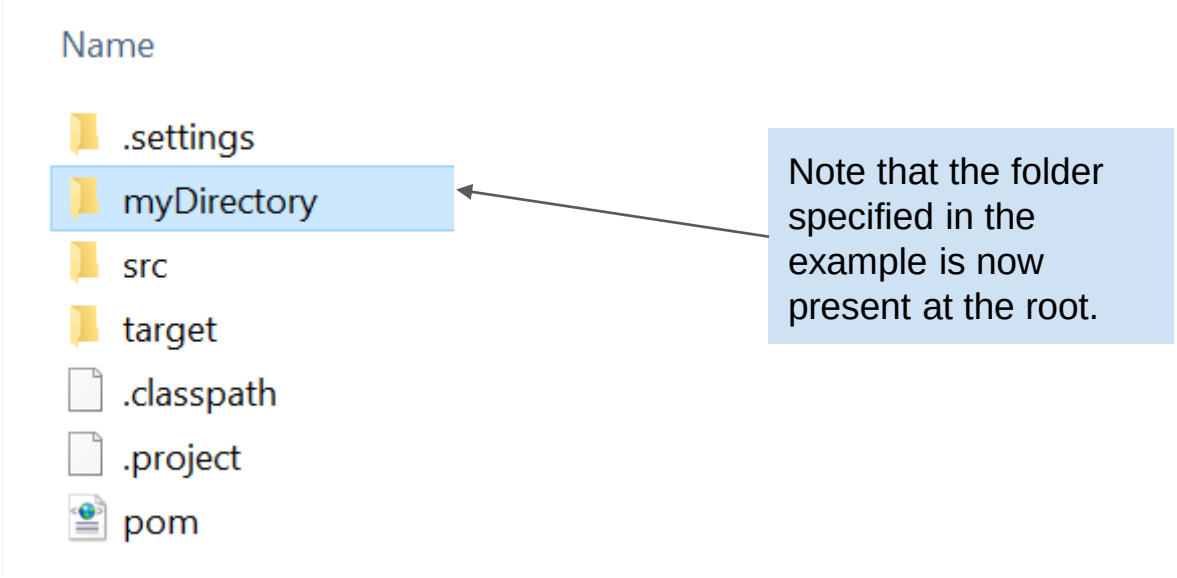
```
public static void main(String[] args) {
   File newDirectory = new File("myDirectory");

   if (newDirectory.exists()) {
     System.out.println("Sorry, " + newDirectory.getAbsolutePath() +
         " already exists.");
   }
   else {
         newDirectory.mkdir();
   }
}
```

We won't create a new directory if it exists.

Otherwise, the .mkdir method will create a new directory.

File class: create a directory.

Just like with reading from files, writing is done with respect to the project root.



File class: create a file.

```
public static void main(String[] args) throws IOException {
   File newFile = new File("myDataFile.txt");
         newFile.createNewFile();
}
```

File class: create a file within a directory.

```
public static void main(String[] args) throws IOException {
   File newFile = new File("myDirectory","myDataFile.txt");
   newFile.createNewFile();
}
```

Writing to a File

Just like with reading data from a file, writing to a file involves bringing in an object of another class. In this case, we will need an instance of the PrintWriter class.

When more than one class is required to solve a problem, we typically refer to these classes as **collaborators**.In this case, the File, and Printwriter classes are collaborators.

Writing a File Example

```
public static void main(String[] args) throws IOException {
   File newFile = new File("myDataFile.txt");
   String message = "Appreciate\nElevate\nParticipate";

   PrintWriter writer = new
        PrintWriter(newFile.getAbsoluteFile());
   writer.print(message);
   writer.flush();
   writer.close();
}
```

Create a new file object.

Create a PrintWriter object.

print the message to the buffer.

flush the buffer's content to the file.

The expected result:

- There will be a new text file in the project root.
- The file will be called myDataFile.txt
- The file will contain each of the three words in its own line.

What is a buffer?

A buffer is like a bucket where the text is initially written to. It is only after we invoke the **.flush()** method that the bucket's contents are transferred to the file.

The flush (and the .close()) can be performed automatically if the the following pattern is used:

```
public static void main(String[] args) throws IOException {
   File newFile = new File("myDataFile.txt");
   String message = "Appreciate\nElevate\nParticipate";

   try(PrintWriter writer = new
      PrintWriter(newFile.getAbsoluteFile())){
         writer.print(message);
   }
}
```

Appending to a File

The previous example regenerates the file's contents from scratch every time it's run. Sometimes, a file might need to be appended to, preserving the existing data content. The PrintWriter supports two constructors:

- **PrintWriter**(**file**), where file is a file object.
- **PrinterWriter**(**outputStream**)
  - outputStream will be an instance of the OutputStream class.
- outputStream constructor can take in 2 arguments, the File object and a boolean value for appending
- FileOutputStream fos = new FileOutputStream(file, true); to append

Appending a File Example

```
public static void main(String[] args) throws IOException {
   File newFile = new File("myDataFile.txt");
   String message = "Appreciate\nElevate\nParticipate";
   PrintWriter writer = null;

// Instantiate the writer object with append functionality.
   if (newFile.exists()) {
       writer = new PrintWriter(new FileOutputStream(newFile, true));
   }
// Instantiate the writer object without append functionality.
   else {
      writer = new PrintWriter(newFile);
   }
   writer.append(message);
   writer.flush();
   writer.close();

}
```

The expected result is that *myDataFile.txt* will be continuously appended with the message text each time it runs.