Module 1-17

File Output

Objectives

- Investigate File and Directory metadata using the File class
- Write text data to a file
- Explain the concept of buffering in File I/O
- 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

```java
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.");
    }
```

File Operations

JavaDoc

exists()	boolean - does the file or directory exist
isDirectory()	boolean - returns true if the path points to a directory
isFile()	boolean - returns true if the path points to a file
getName()	Returns the name of the file or directory
getAbsolutePath()	Returns the absolute path of the file or directory
mkdir()	Creates a new directory in the specified location
createNewFile()	Creates a new File in the specified location

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 2).
- Send a HTML view back to the user (Module 3).

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

File class: create a directory.

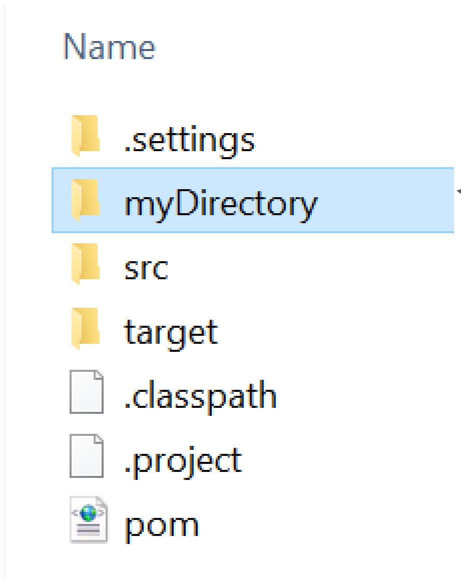
```
public static void main(String[] args) {
   File newDirectory = new File("myDirectory");

   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.


Name

.settings
myDirectory
src
target
.classpath
.project
pom

Note that the folder specified in the example is now present at the 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

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

    PrintWriter writer = new PrintWriter(newFile);
    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.

PrintWriter

`java.io.PrintWriter`

JavaDoc

```
File file = new File(pathToFile);


try(PrintWriter writer = new PrintWriter(file))
{
        writer.println(text);
}
```

The File is referenced by a File object using the path to the File.

The File object is then passed as an argument to the PrintWriter, which opens it for writing.

While the File is open the printWriter can then write data to the file by pushing it onto the output stream.

The connection to the File must be closed once writing is complete, so it is important to use a try-with-resource, which allows Java to auto-close the file once the PrintWriter is out of scope.

What is a buffer?

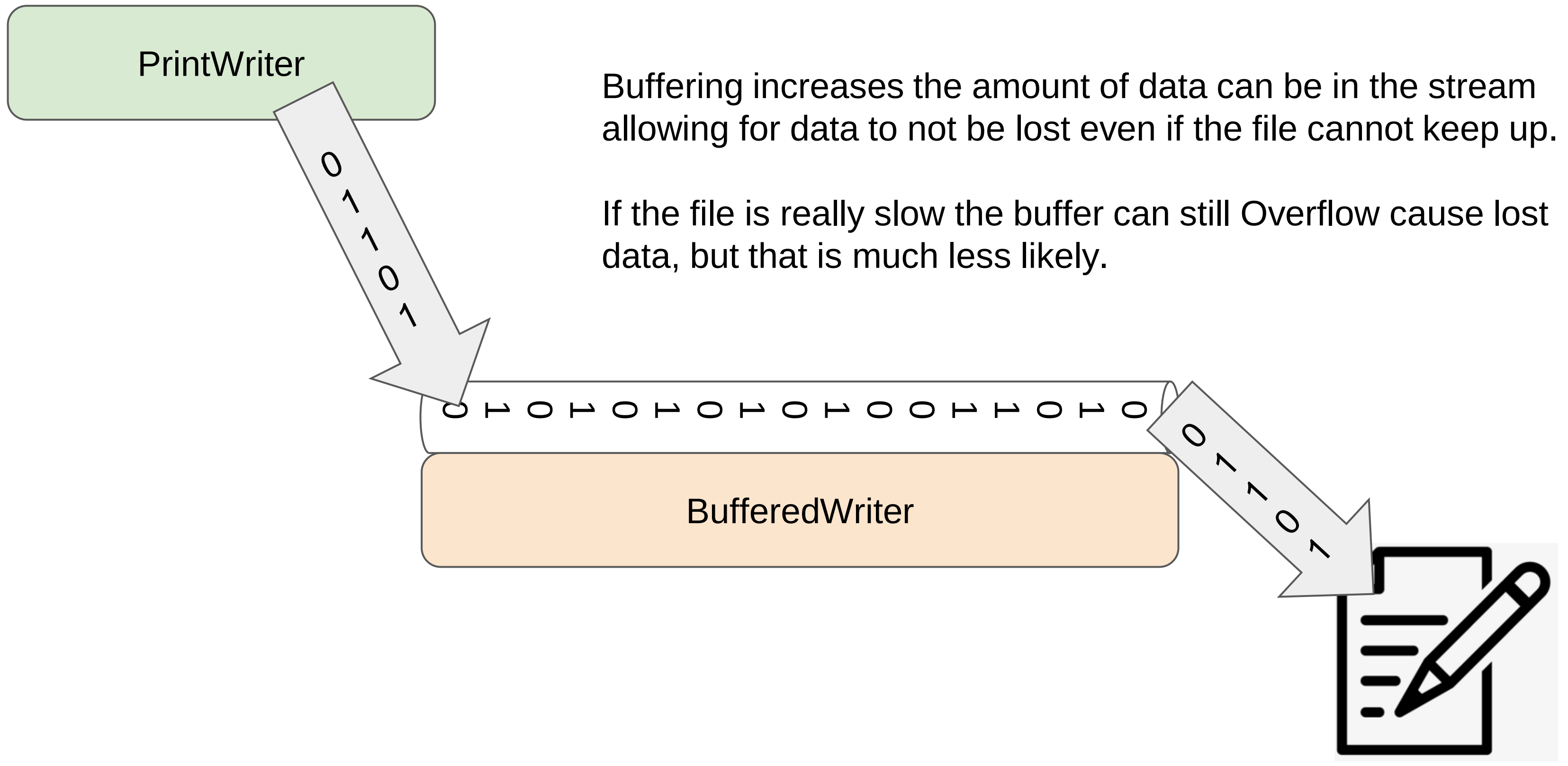
A buffer is like a bucket where the text is initially written to. It is only after we invoke the **.flush()** method that the bucket's contents are transferred to the file.

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)){
         writer.print(message);
   }
}
```

Try with resources…

Buffering



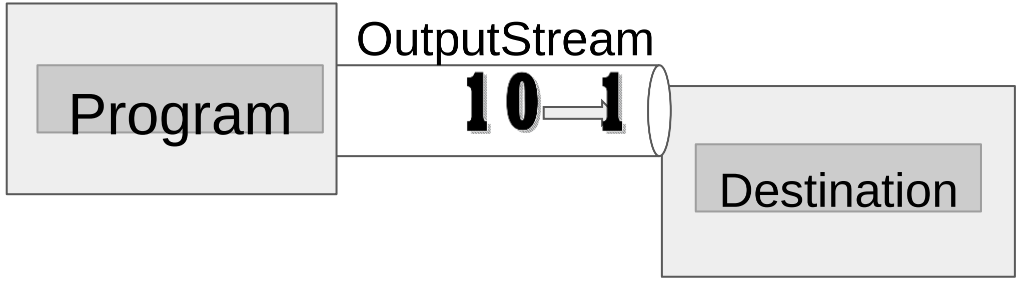
Buffering increases the amount of data can be in the stream allowing for data to not be lost even if the file cannot keep up.

If the file is really slow the buffer can still Overflow cause lost data, but that is much less likely.

Flushing a Stream

Sometimes other operations occurring on the system or in the application can cause data to remain in the stream without an opportunity to write it to the destination.

Flushing forces all the data in the stream to be immediately sent from the Stream to the destination.



Streams have a flush() method that can be called to immediately flush them.

System.out.flush()
PrintWriter.flush()
BufferedWriter.flush()

A stream should never be closed without first being flushed. If the stream is opened with a try-with-resource, it will automatically flush the stream before closing it.

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.
- **PrintWriter**(**fileWriter**)
  - fileWriter will be an instance of the FileWriter class.
- FileWriter constructor can take in 2 arguments, the File object and a boolean value for appending
- FileWriter fw = new FileWriter(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 FileWriter(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.