# CIS 3145 Class Notes: Text Chapter 17

## Input / Output (I/O): Text and Binary File Processing

**Objectives**

* Describe the java.nio.file package path and file class
* Describe files and input/ output (I/O) streams
* Create code for I/O exceptions
* Write code to read from and write to text files
* Write code to read from and write to binary files

**Java.nio.file package**

An example of a file **directory path is “***c:/murach/java/files*”, while the **file** in a directory has a name with an extension “*products.txt*”.

To **read** from and **write** to files we first find them on the computer. The **Path** class uses static methods to define the file **name** and **location** on a computer. The **file** and **directory** are each name elements for the **path** object.

The **File** class represents the file in the directory as an object. It also uses static methods to test properties of paths objects. We must **dynamically** determine if files and paths exist before trying to manipulate them. The methods also allow us to create files and directories if they don’t exist, as well as delete files and directories.

(1) Create a **path** object with the static Paths.get() method to represent the **directory**.

The **dirPath** is an object of the **Path** object type.

Path dirPath = Paths.get (*"C:/murach/java\_netbeans/files"*);

(2) Create a **path** object with the static Paths.get() method to represent the **file’s** location in a directory.

Path filePath = Paths.get(*"C:/murach/java\_netbeans/files", "test.txt")*

(3) Create a **file** object with the Path object’s **toFile**() method to represent the **file** itself.

File myFile = filePath.toFile();

The **Files** class has **static methods** to **create** and **delete** files and directories on the computer. The **File** class also has methods to test the properties of the files and directories: does it **exist**, is it **readable** or **writeable**, and what is the **size** of the file.

**Two type of files / Two types of operations**

There are two types of files used to store data in computers: **text** and **binary**. **Text** files use standard 16-bit UNICODE ([ASCII](https://en.wikipedia.org/wiki/ASCII) as a subset of UNICODE) characters. As such we can read a file when opened in Notepad or other word processing program. Storing a number in a text file means that the number one is stored as the character “1”. Since a single character is 2 Bytes this is a very wasteful way to store a number. **Binary** files are a more efficient way to store data and are used when humans do not have to read the file directly.

From the view point of the **program** data is taken into the program (**Input**) or exported out of the program (**Output**). Thus reading and writing for a program are associated with input/output, aka I/O.

**Streams: the middleman of IO**

For a java program to process a file it must create a stream object. The stream object will act as a buffer between the program and the file. Writing to a file requires an *output stream* object, while reading from the file requires an *input stream* object.

There are separate classes of streams for ***binary*** and ***character*** files.

* Binary files are smaller and a more efficient way to store data

Program 🡪 Buffer -> Stream 🡪 File

**Buffered Streams** are more efficient. We read/write a **block** of data instead of one bit at a time.

The code to **write** (**output**) to a file starts by creating an object that represents the file (or using a string to represent the file location and name). This describes how to write to a **text** file. Because I/O operations can fail when the hardware they are reading and writing to fails (a hard drives crashes or the network goes down), these operations must use try…catch blocks to catch possible **run-time** errors.

* The **FileWriter** class is used to create an object that represents a **stream** to a file. It can take the **file object** as input argument.

FileWriter **myFileWriter** = new FileWriter(**myFile**);

* The **BufferedWriter** class creates an object that represents a buffer connected to a stream. The **stream** object is used as an input argument to create the buffer object.

BufferedWriter **myBuffer** = new BufferedWriter(**myFileWriter**);

* The **PrintWriter** class is used to create an object that represents the file and the operations we can perform to write to the file. The **buffer** object is used as an input argument to create this object.

PrintWriter **myWriter** = new PrintWriter(**myBuffer**);

* The **PrintWriter** object uses the **println**() or **print**() methods to write to the file. Always remember to use the **close**() method to close the connection to the file. Closing will flush the data out of the stream into the file. Otherwise there might not be data in the file.

**myWriter**.println("test the output.");

**myWriter**.close();

The code to **read** (**input**) from a **text** file matches the writing process and starts with a file object (or string that represents the file location and name).

* The **FileReader** class is used to create an object that represents a **stream** to a file. It can take the **file object** as input.

FileReader **myFileReader** = new FileReader(**myFile**);

* The **BufferedReader** class creates an object that represents a buffer connected to a stream. The **stream** object is used as input to create the buffer object.

BufferedReader **myReader** = new BufferedReader(**myFileReader**);

* The **BufferedReader** object uses the **read**() and **readLine**() to read data from the file.

String Message = **myReader**.readLine();

Any IO operation has the potential of causing a **run time error** because every hard drive has the potential of crashing. The resulting exception can be **thrown** or **caught** (see chapter 16). If an exception is thrown and not caught when it is first created, the code that calls the error must catch the error and report it to the user. Uncaught errors have the potential of causing the program to crash, thus this must be avoided at all costs.