Java Chapter 13 – File Input and Output

**UNDERSTANDING COMPUTER FILES:**

Data items can be stored in 2 broad types of storage devices in a computer system

* Volatile Storage:
* Temporary; volatile values (like those stored in variables) are lost when a computer loses power
* \*Writing a Java program that stores a value in a variable
* \*RAM – the temporary storage w/in a computer
* Nonvolatile Storage:
* Permanent; not lost when a computer loses power
* \*Writing a Java program & saving it to a disk
* Computer File 🡪 collection of data stored on a nonvolatile device
* Files exist on Permanent Storage Devices (
* Categorized by the way they store data
* Text Files

-some are Data files

-some are Program/Application files

* Binary Files

\*Common Characteristics of Files:

* Size – specifies the space it occupies on a storage device
* Name
* Specific time of creation

**PATH & FILE CLASSES:**

\*To use both classes in your Java program 🡪 import java.nio.file\*;

* **Path Class:** used to create objects that contain info about files & directories such as their locations, sizes, creation dates, & whether they even exist
* **Files Class:** used to perform operations on files & directories such a deleting them, determining their attributes, & creating input/ output streams

Creating a Path:

* Determine the default file system on the host computer by creating a FileSystem object & use the getDefault() method in the FileSystems class

FileSystem fs = FileSystems.getDefault();

* FileSystem 🡪 used to instantiate the object
* FileSystem***s*** 🡪 contains Factory Methods which assist in object creation
* Define a Path:
* Use the getPath() method w/ the FileSystem object

Path path = fs.getPath(“C;\\Java\\Chapter13\\Data.txt”);

* To enter a backslash as a path delimiter w/in a string 🡪 must type 2 backslashes
* Alternative 🡪 FileSystem method getSeparator(): returns operator for current OS

Another Way to Create a Path:

* Use the **Path*s* Class 🡪** a helper class that eliminates the need to create a FileSystem object:
* get() method: calls the getPath() method of the default files system w/out requiring you to instantiate a FileSystem object

Path filePath = Paths.get(“C:\\Java\\Chapter13\\SampleFiles.txt”);

* Use the Path identifier (filePath) to refer to the file & perform operations on it:
* C:\\Java\\Chapter13\\SampleFiles.txt 🡪 file name when referred to by the OS
* filePath 🡪 file name when referred to by/ within the application
* Every Path is either Absolute or Relative
* When working w/ a path that contains only a filename, the file is assumed to be in the same folder as the program using it

Retrieving Information About a Path:

\*Path Class Methods:

* String toString():
* Path getFileName():
* Path getName(int):

\*A Path’s elements are accessed using an index:

🡪 top-level element in the directory structure – index 0

🡪 lowest element in the structure –

* index of 1 less than the number of items on the list
* the structure accessed by this method
* returns the name in the position of the path specified by the integer
* int getNameCount():

Converting a Relative Path to an Absolute Path:

* toAbsolutePath(): converts a relative path to an absolute one

Checking File Accessibility:

* checkAccess():
* verifies that a file exists and that the program can access it as needed

filePath.getFileSystem().provider().CheckAccess();

* import static java.nio.file.AccessMode.\*;
* allows you to access constants that can be used as arguments to the method
* Can use any of the following (& multiple) arguments to the checkAccess() method:
* No Argument – checks that the file exists

\*Alternative 🡪 use the Files.exist() method & pass it a Path argument

* READ – checks that the file exists & that the program has permission to read the file
* WRITE – checks that the file exists & that the program has permission to write the file
* EXECUTE – checks that the file exists & that the program has permission to execute the file

\*If the file named in the method call cannot be accessed – an IOException is thrown

🡪 Must import the java.io.IOException package

Deleting a Path:

* Files class method delete():
* Accepts a Path parameter & deletes the last element (file/directory) in a path
* Throws an exception of deletion fails:
* Trying to delete a file that doesn’t exist 🡪 throws a NoSuchFileException
* Attempting to delete a directory that isn’t empty 🡪 DirectoryNotEmpyException
* Attempting to delete a file you don’t have permission to 🡪 SecurityException
* Other input/output errors 🡪 IOException
* Files class method deleteIfExists(): if file doesn’t exists – no exception is thrown

Determining File Attributes:

* Files class method readAttributes():
* Used to retrieve useful info about a file
* Returns an instance of the BasicFileAttributes class
* Takes 2 arguments 🡪 a Path argument & BasicFileAttributes.class

BasicFileAttributes attr = Files.readAttributes(filePath, BasicFileAttributes.class);

* Use various methods for retrieving file info w/ the created BasicFileAttributes object:
* size(): returns the size of a file in bytes
* Time Methods:

🡪 Each return a FileTime object

🡪 FileTime objects are represented in the format – yyyy-mm-ddThh:mm:ss

* creationTime() & lastModifiedTime()
* \*compareTo(): determines the time relationships between files

FILE ORGANIZATION, STREAMS, & BUFFERS:

\*Data stored in variables w/in programs is temporary

\*Variables are stored in the computer’s main or primary memory (RAM)

\*To retain data 🡪 must save the data on a permanent, secondary storage device (like a disk)

\*Businesses organize data in a hierarchy:

* Character:
* The smallest useful piece of data to most users
* Can be any letter, number, or other special symbol that comprises data
* Made up of bits;

🡪 users usually concerned w/ the meaning rather than the internal representation

* Can be any group of bits – does not necessarily represent a letter or number
* Not necessarily created w/ a single keystroke
* \*Think of it as a unit of info rather than data w/ a particular appearance\*
* Characters are grouped into fields
* Field:
* Group of characters that has some meaning

ex: the characters ‘T’ ‘o’ ‘m’ represent the data field of the first name ‘Tom’

* Fields are grouped together to form records
* Records:
* A collection of fields that contain data about an entity
* \*The data fields stored in Java classes can be thought of as a record

ex: a person’s first & last name, SSN, & zip code represent that person’s record

* Records are grouped to create files
* Files:
* Data files consist of related records

ex: A companies personnel file that contains one record for each employee

* Can be used as a Sequential Access File when each record is accessed in the order it was stored
* Usually each record is stored in order based on the value in some field; SSN, item #
* When records are not used in sequence – file is used as a random access file

\*When records are stored a data file – their fields can be organized one to a line, or a character can be used to separate them (Like a file of CSV)

* Before an application can use a data file – must open the file:
* Java apps do this by creating an object & associating a stream of bytes w/ it
* \*Failing to close an input file (file from which you are reading) – no serious effects
* When you finish using a file – should close the file:
* Meaning make it no longer available to your application
* \*Failing to close an output file (file to which you are writing) – the data might become inaccessible

\*Should always close every file you open, & as soon as you no longer need it

- leaving it open 🡪 uses computer resources & w/in a network another program might be waiting to use it

\*Java views a file as a series of bytes\*

\*when performing an input operation in an app – bytes flow into the program from an input device, through a Stream

\*when performing an output operation – some bytes flow out of the app through another Stream to an output device

* Most stream flow in only one direction
* \*Random Access Files use streams that flow in two directions
* Input & output operations are usually the slowest in any computerized system because of the limitations imposed by the hardware
* Because of this – professional programs often employ buffers

( StringBuilder object sets aside a memory block called a buffer

* Memory location where bytes are held after they are logically output but before they are sent to the output device )

\*improves program performance – sometimes flush it before closing it

**JAVA’S INPUT OUTPUT CLASSES:**

\*Some Java classes used for IO operations

* InputStream, OutputStream, & Reader – subclasses of the Object class (all abstract)

\*Abstract classes contain methods that must be overwritten in their child classes\*

Object –

* **InputStream**
* FileInputStream
* FilterInputStream
* BufferedInputStream
* **OutputStream**
* FileOutputStream
* FilterOutputStream
* BufferedOutputStream
* PrintStream
* **Reader**
* BufferedReader
* BufferedWriter

\*Java’s System class contains a PrintStream object named System.out – print() & println() methods

* Systems class also defines a PrintStream object named System.err
* Output from System.err & System.out can go on the same device
* System.err 🡪 error messages
* System.out 🡪 valid output

\*Usually no need to – but can create an OutputStream object & assign System.out to it

FIGURE 13-15

* getBytes(): converts the String to an array of bytes
* write(): accepts the byte array and sends it to the output device

Writing to a File: (OutputStream)

* Other output devices can be assigned to OutputStream references, allowing your applications to save data to them
* Instead of assigning the standard output device to OutputStream, you can assign a file
* Java lets you assign a file to a Stream object so that screen output & file output work in exactly the same manner
* Can create a writable file by using the Files class newOutputStream() method:
* Creates a file if not already existing, opens the file for writing, & returns an OutputStream that can be used to write bytes to the file

Reading From a File: (InputStream)

\*Can also create an InputStream, assign System.in to it & use the class’s read() method w/ the created object to retrieve keyboard input

🡪 More efficient

🡪 use Scanner class for keyboard input

🡪 use InputStream class to input data that has been stored in a file

* To open a file for reading:
* Use the Files class newInputStream() method:
* Accepts a path parameter & returns a stream that can read bytes from a file
* BufferReader Class: reads a line of text from a character-input stream, buffering characters so that reading is more efficient

**SEQUENCIAL DATA:**

FIGURE 13-21?

* BufferWriter Class: writes text to an output stream, buffering the characters
* Has 3 overloaded write() methods

Lowkey skipped the rest of this section

**RANDOM ACCESS FILES:**

\*Businesses store data in sequential order when they use the records for batch processing

\*For many applications, sequential access is inefficient

🡪 real-time applications 🡪interactive programs (users make direct requests)

* RAF:
* Files in which records can be retrieved directly in any order (AKA direct/instant access files)
* Use FileChannel class to create your own RAF’s:
* File channel object 🡪 avenue for reading & writing a file
* A file channel is Seekable 🡪 can search for a specific file location & operations can start at any specified position
* Several of the FileChannel methods use a ByteBuffer:
* Simply a holding place for bytes waiting to be read or written
* An Array of bytes can be Wrapped (encompassed) into a ByteBuffer using the ByteBuffer wrap() method:
* Wrapping a bye array into a buffer causes changes made to the buffer to change the array, & causes changes made to the array to change the buffer
* Creating a usable FileChannel for randomly writing data requires creating a ByteBuffer & several other steps:

FIGURE 13-28 FOR SURE

WRITING RECORDS TO A RANDOM ACCESS DATA FILE:

\*When you store records in a file 🡪 more useful to be able to access the eighth or 12th record rather than the eighth or 12th byte

🡪 Multiply each record’s size by the position you want to access

* Access the nth record in a FileChannel named fc using:

fc.position((n-1) \* 50);

* Writing a RAF:
* Place records into the file based on a key field
* 1st step 🡪 create a file that holds default records (0’s for ID’s & pay, blanks for names)

READING RECORDS FROM A RANDOM ACCESS FILE:

\*Just because a file is created as a RAF, does not mean it has to be treated as one

\*You can process/access a RAF either Sequentially or Randomly

* Sequentially:
* Application that reads through the record files sequentially in a while loop
* Program only displays a record when the ID is not the previously set default value

\*If you simply want to display records in order based on their key fields, there’s no need to create a RAF & waste unneeded storage. Instead:

* Randomly:
* Instead – sort the records using a technique from the Arrays/Sorting Algorithms chapter