Fundamentals of Programming 2

Lecture 11

Aurelia Power, TU Dublin - Blanchardstown Campus, 2019

*** Notes based on the Java Oracle Tutorials (2018), Deitel & Deitel (2015), and Horstman (2013)**

Files Recap

- **Files** are means of storing data for long term retention -> data saved to files is persistent data
- **Text files** store ASCII values and are accessed sequentially; they are organised into a sequence of lines separated by a line separator.
- They are read or written character by character, word by word, or line by line; They are meant to be read by us/text editors
- **Binary files** store data in binary format and they cannot be read by us/text editors;
- They can be accessed sequentially, but also random.
- In java, support for files and file processing is found in **java.io package**
- The **File** class is used to create objects that represent files but they do NOT open files/streams, or read or write data
- The File class has **several overloaded constructors** (check the API …);
- The most common constructor takes in a **String** that specifies the path and the name of the file:

```
File file = new File("C:\\users\\someFile.txt");
```

- The file class has many useful methods…check API
- NOTE: make sure that you escape the escaping character: \\

```java
import java.io.File;

/**
 * @author Aurelia Power
 *
 */
public class Ex3 {
    public static void main(String[] args) {
        //create a file object using the given path: your path may be different
        File file = new File("C:\\users\\FOP2\\integers.txt");
        //check to see if the file exists
        //should print true because you have created manually that file in exercise 2
        System.out.println(file.exists());
        //check to see if it is a directory/folder
        //should print false because it is not a directory
        System.out.println(file.isDirectory());
        //retrieve the path of the file
        //should print different path for each one of you
        System.out.println(file.getPath());
        //set permission to modify or write to that file
        file.setWritable(true);
        //because we set writable to true, the canWrite method
        //returns true, so it prints true
        System.out.println(file.canWrite());
    }//end main
}//end class
```

Reading Text Files

Character Streams are suited for read/write operations with text files; also called readers and writers, respectively.

To read a character/tex-based file we used **FileReader** and **BufferedReader** classes;

FileReader class is designed especially to read character files or streams of characters.

-You can use several **read** methods from the FileReader class to read data from a character file, but they are inefficient

-So … it is recommended that we use a buffer that holds the bytes from the stream and can be accessed them as needed, without having to re-read the entire stream again and again;

-For this purpose we will use the **BufferedReader** class to wrap it around the FileReader.

```
File file = new File(“someFile.txt”);
FileReader reader = new FileReader (file);
BufferedReader bufferR = new BufferedReader(reader);
```

OR

```
BufferedReader bufferR = new BufferedReader(new FileReader (new File(“someFile.txt”)));
```

Writing to Text Files

- To write to a text file we used the **FileWriter and BufferedWriter** classes;
- **FileWriter** class is designed especially to write to a character/text file.
- You can use several **write** methods from the FileWriter class to write data to a character file, but they are inefficient.
- So, it is recommended that we use a buffer: for this purpose, we use a **BufferedWriter** object to wrap it around the FileWriter object..

```
File file = new File("someFile.txt");
FileWriter writer = new FileWriter (file);
BufferedWriter bufferW = new BufferedWriter(writer);
```

OR

```
BufferedWriter bufferW = new BufferedWriter(new FileWriter (new
    File("someFile.txt")));
```

```java
/**
 * @param path
 * prints every second line from file
 * */
public static void printEvery2ndLine(String path) {
    /* open the input stream for reading the text file
     with the given path and wrap a buffer around it
    for more efficient reading operations */
    try(BufferedReader buffer = new BufferedReader(
            new FileReader( new File(path) ))){
        /* call the method readLine twice in each iteration
        because not only reads and returns the contents of the current
        line, but also moves the pointer to the next line;
        only print what it returns the 2nd time if it's not null!!! */
        while(buffer.readLine() != null) {
            String line = buffer.readLine();
            System.out.println((line != null) ? line : "");
        }
    }catch(IOException e) {
        e.printStackTrace();
    }
}// end of printEvery2ndLine
```

```java
/**
 * @param path, sents
 * writes to the file each sent on a separate line
 * */
public static void writeSentsToFile(String path, ArrayList<String> sents) {
    /* open the output stream for to the text file
     with the given path and wrap a buffer around it
    for more efficient writing operations */
    try(BufferedWriter buffer = new BufferedWriter(
            new FileWriter(new File(path)))){
        /* go through string element of the array list and
         write to the file using the buffer object */
        for(String sent: sents) {
            buffer.write(sent + "\n");
            //can also invoke newLine()
        }
    }catch(IOException e) {
        e.printStackTrace();
    }
}// writeSentsToFile
```

```
I love Maths
I love Programming
I love Data Science
```

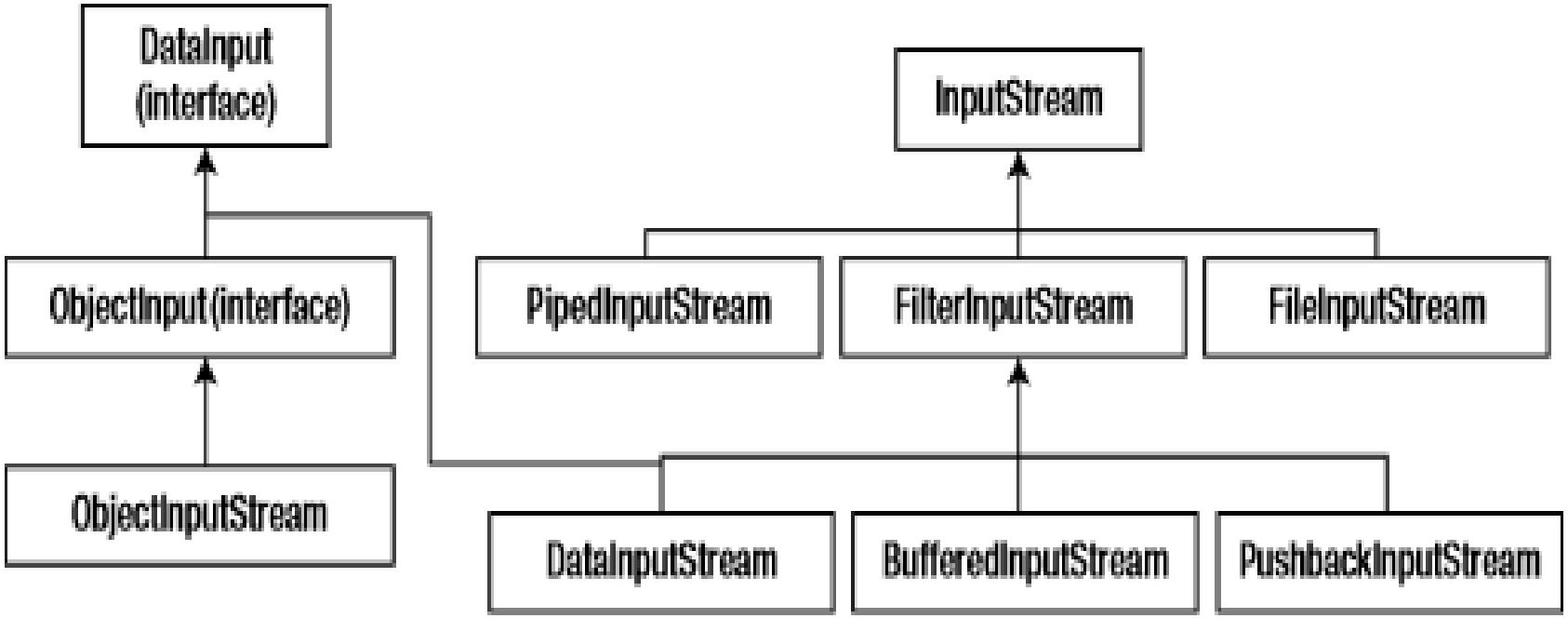
```java
/**
 * @param args
 * testing our 2 methods...
 */
public static void main(String[] args) {
    // the name and the path of the file
    String path = "sents.txt";
    // create a list to hold the sentences
    ArrayList<String> sents = new ArrayList<>();
    sents.add("I love studying");
    sents.add("I love Maths");
    sents.add("I love cleaning");
    sents.add("I love Programming");
    sents.add("I love waking up in the morning");
    sents.add("I love Data Science");
    sents.add("I love rain");
    // write the sentences in the list to a text file
    writeSentsToFile(path, sents);
    // read only the second line from the text file
    printEvery2ndLine(path);
    //.... use other lists and paths to test it further
}// end main
```

Today … Byte Streams and Binary Files

- **Byte Streams** are suited for read/write operations with binary files;
- Support is provided also in **java.io package**
- Typically, we use the **InputStream** and **OutputStream** abstract classes and their respective subclasses.
- We also deal with situations that represent input/output exceptions, such as missing files, so we need to ensure that we have a mechanism that will allow us to continue execution of the program:
- Again, we will use **try-with-resources-catch mechanism** to deal with **IOExceptions.**

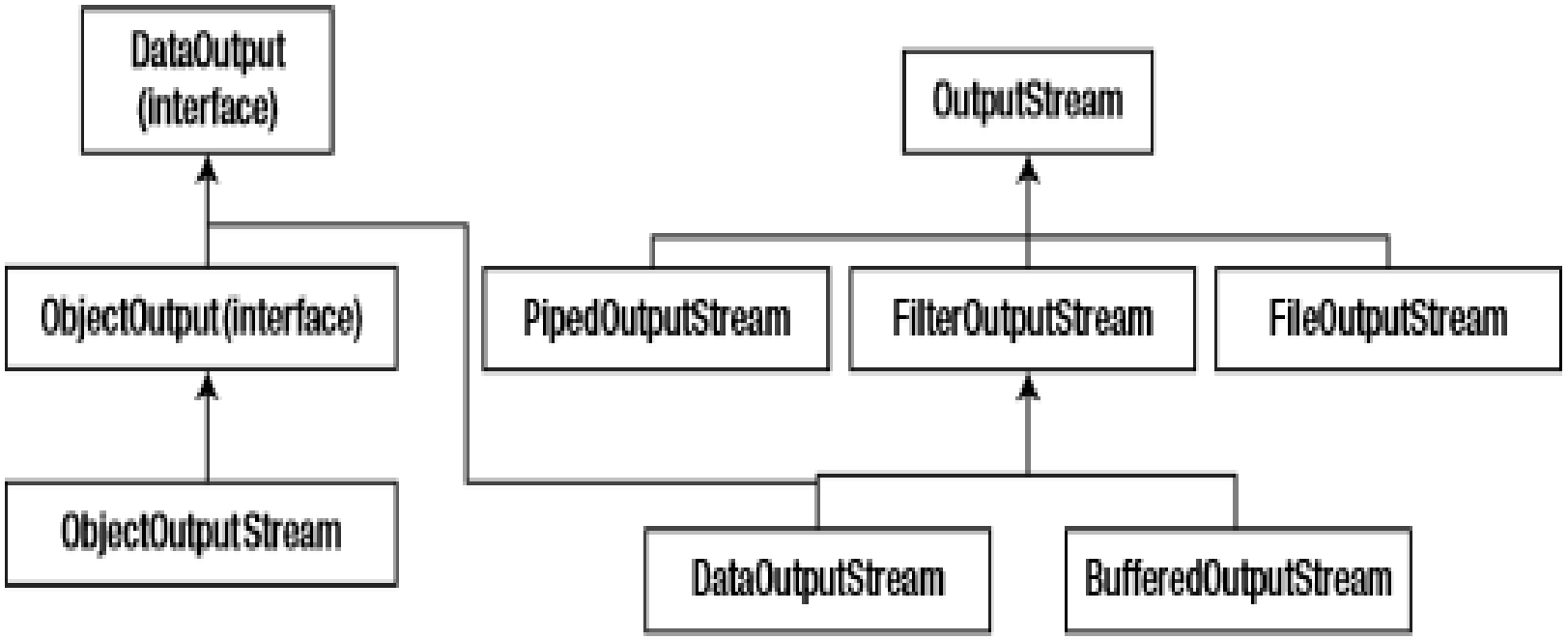
Byte Streams

- The hierarchy of classes derived from **InputStream**:



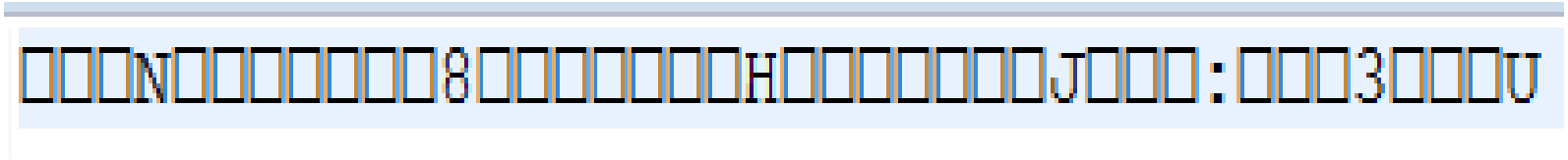
Byte Streams

- The hierarchy of classes derived from **OutputStream**:



Reading a binary file sequentially

- Example: **intList.dat** - contains int values
- If you try to open it, you get something like the following output:



- So we need to use an **appropriate application** to read such files;
- We can use the **FileInputStream** which is designed to read the raw bytes from a file, such as an image file; these raw bytes must then be converted so to be read by us or text editors;

Reading a binary file sequentially

- To do the conversion efficiently, we use the **DataInputStream** to wrap it around an input stream to read primitive data types such characters, integers, etc., and strings

- **DataInputStream** has methods to read all **primitive data types and strings**: readInt(), readDouble(), readBoolean(), readChar(), readUTF(), etc. (check API);

- Like with readers, if the file does not exist on your machine, an IOException will be thrown, but it will be caught by the catch clause, in which case the statements in the catch block will be executed.

Reading a binary file that contains characters sequentially

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

/**
 * @author Aurelia Power
 *
 */
public class ReadBinaryFileSequentially {
    public static void main(String[] args) {
        /* use a try-with-resources to open the file and the read stream;
        this will ensure that they are closed automatically when finsished*/
        try(FileInputStream input = new FileInputStream("charList.dat");
                DataInputStream data = new DataInputStream(input);){
            /*as long as there is data available in the data stream,
            read it using the corresponding method for that data type,
            in our case char, and print it to console using print statement */
            while(data.available() > 0){
                System.out.print(data.readChar() + " ");
            }
        }catch(IOException ioe){
            ioe.printStackTrace();
        }
    }//end main
}//end class
```

Reading a binary file that contains doubles sequentially

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

/**
 * @author Aurelia Power
 *
 */
public class ReadBinaryFileSequentially2 {
    public static void main(String[] args) {
        /* use a try-with-resources to open the file and the read stream;
        this will ensure that they are closed automatically when finsished*/
        try(FileInputStream input = new FileInputStream("doubles.dat");
                DataInputStream data = new DataInputStream(input);){
            /*as long as there is data available in the data stream,
            read it using the corresponding method for that data type,
            in this case double, and print it to console using print statement */
            while(data.available() > 0){
                System.out.print(data.readDouble() + " ");
            }
        }catch(IOException ioe){
            ioe.printStackTrace();
        }
    }//end main
}//end class
```

Reading Binary Files Sequentially – step by step…

1. **Create a file input stream** by passing to its constructor as argument the name of the file (there are other constructors … check API);
2. **Create the data input stream** that takes as argument the file input stream created in step 1;
3. Then use a while loop **to read one value** at a time, by invoking the method **readChar()** or **readDouble(), or readInt(), etc.,** inside the loop body, depending on what type of data the file contains; in the loop, we also **display** what is read from the file.
4. We then ensure that, in case an exceptional situation occurs, we provide a way to **catch it**, and print an appropriate error message.

NOTE1: the boolean condition of the while loop is provided by comparing what the **available()** method returns (an int value representing the remaining number of bytes in the data stream) to 0: if the number of bytes is greater than 0, the boolean resolves to true, otherwise to false.

NOTE2: because we used try-with-resource, we do not need to explicitly **close the streams**.

Writing to a binary file sequentially

- Example: we want to write some integers this time to a file **ints.dat** - which will contains integer values;
- We need to use an **appropriate application** to write them;
- We can use the **FileOutputStream** class which is designed to write raw bytes to a file (such as an image file).
- But, to write efficiently primitive data types and strings to a binary file, we need to wrap the output stream into a **DataOutputStream** ;
- The DataOutputStream provides methods to write all **primitive data types and strings**: writeInt(), writeDouble(), writeBoolean(), writeChars(), writeUTF(), etc. (check API);
- If the file to which we want to write does not already exist on your computer, it will be automatically created.

Writing integers to a binary file sequentially

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * @author Aurelia Power
 *
 */
public class WriteToBinaryFileSequentially {
    public static void main(String[] args) {
        /*use a try-with-resources to open the resources; if the file does
         *  not already exist, it will be automaticaclly created */
        try(FileOutputStream out = new FileOutputStream("ints.dat");
                DataOutputStream data = new DataOutputStream(out);){
            //declare the array of integers
            int[] ints = {7, 34, 25, 1, 16, 10, 19};
            /* use a for loop to access each element of the array and write it
             * to the file by invoking the writeInt method */
            for(int i = 0; i < ints.length; i++){
                data.writeInt(ints[i]);
            }
        }catch(IOException ioe){
            ioe.printStackTrace();
        }
    }//end main
}//end class
```

Writing characters to a binary file sequentially

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Random;

/**
 * @author Aurelia Power
 *
 */
public class WriteToBinaryFileSequentially2 {
    public static void main(String[] args) {
        /*use a try-with-resources to open the resources; if the file does
         *  not already exist, it will be automatically created */
        try(FileOutputStream out = new FileOutputStream("chars.dat");
                DataOutputStream data = new DataOutputStream(out);){
            //declare a Random object that will allow us to create random integers
            Random rand = new Random();
            /* generate 10 random integers between 32 and 126 and then convert them
             * to char (using the cast operator; rememeber the ASCII table contains
             * characters represented as positive integers between 32 and 126); then
             * write each char to the file invoking the method writeChar */
            for(int i = 0; i < 10; i++){
                char c = (char)(rand.nextInt(95)+32);//127 is not included
                data.writeChar(c);
            }
        }catch(IOException ioe){
            ioe.printStackTrace();
        }
    }//end main
}//end class
```

Writing Binary Files Sequentially – step by step…

1. **Create a file output stream** by passing to its constructor as argument the name of the file (there are other overloaded constructors … check API);
2. **Create the data output stream** that takes as argument the file output stream created in step 1;
3. Then use a for loop **to write each datum** by invoking the method **writeInt(), or writeChar()**, etc. depending on the data type that you want to write to the file.
4. We then ensure that, in case an exceptional situation occurs, we provide a way to **catch it**, and print an appropriate error message, or use the printStackTrace() method inherited from the Exception class.

NOTE: because we used try-with-resource, we do not need to explicitly **close the streams**; however, if we were to use a traditional try-catch mechanism, we would have to also close the resource in a finally clause.

Quiz

- You can read an image file using an output stream. True or false??

False – you need an input stream for reading

- If you have code that has the potential of throwing an IOException, but you haven't dealt with it, when you run it can end with an exception

False – your code in fact won't compile: you must deal with IOException

- Consider the following code (assume imports, declarations, try-catch, and streams open appropriately):

```
int k = 7;
while(k <= 10){
    outputStream.writeDouble(++k);
}
```

- Does it compile??

Yes, because an integer fits into a double.

- What will it output??

Nothing. There are no output statements

- What will be the contents of the file, assuming that the file had no data until the writeDouble method was invoked?

The following double values: 8.0, 9.0, 10.0, 11.0 (note that there is no commas or spaces between the values in the actual file).

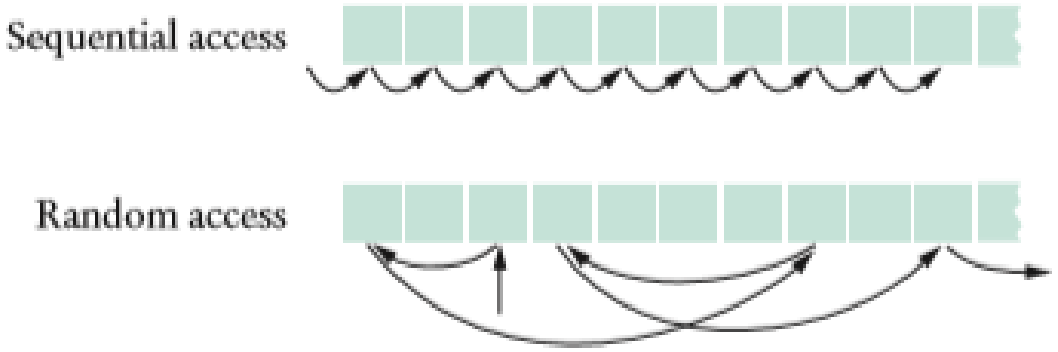
Random Access Binary Files

- **Sequential binary files** are read in sequence from start to finish…so there are some **drawbacks** :
- New data can only be inserted by appending them to the file at the end;
- If you want to change some data in the file, you are required to process all data in sequence, change them and write them to a new file, then delete the old file, and rename the new file to the new file →a very expensive process.

- To overcome these problems we can use **random access binary files** which are represented as arrays of bytes (indexing of bytes starts at 0)
- We use the **RandomAccessFile class** (also in java.io package);

Random Access Binary Files

- **Random access files** allow us to access bytes at arbitrary locations without having to read all the previous bytes (as it is the case of sequential access).



- Random access files also allow us to make changes as we access the contents of the file;
- It maintains a **file pointer** that is advanced to the next byte as the file is read or written to ;

Random Access Binary Files

- They can be open typically in 4 modes (r, rw, rws, rwd), but we focus on: **read-only** (r) or **read-write** (rw), using one of the 2 overloaded constructors: one that takes 2 Strings, the other that takes a File and a String.

```
RandomAccessFile raf = new
                    RandomAccessFile("intList.dat", "rw");
RandomAccessFile raf2 = new
                        RandomAccessFile("intList.dat",  "r");
```

OR

```
RandomAccessFile raf = new
                    RandomAccessFile(file1, "rw");
RandomAccessFile raf2 = new
                        RandomAccessFile(file2, "r");
```

- We can access the length of the file representing its size in bytes using the length method:

```
long fileLength = raf.length();
```

Random Access Binary Files

- The file pointer is **obtained** by the **getFilePointer** method as a long value:

  ```
  long filePointer = raf.getFilePointer();
  ```

- We can set the file pointer using the **seek** method which takes a long value:

  ```
  raf.seek(someLongValue);
  ```

- The read and write operations/methods both move the pointer;
- **Read operations** can be carried out for all primitives and Strings; for example, the RandomAccessFile class has readInt(), readLong(), readDouble(), readBoolean(), readChar(), readUTF(), etc.
- Similarly, **write operations** can be carried out for all primitives and String; for example, the RandomAccessFile class has writeInt(), writeLong(), writeDouble(), writeBoolean(), writeChar(), writeUTF(), etc.
- Like with the readers and input streams, if you try to read a file using RandomAccessFile that does not exist on your machine, it will throw an exception, but it should be caught.
- Like with writers and output streams, if you try to write to a file that does not exist on your machine using RandomAccessFile, the program will create it automatically.

Random Access Files – method for reading a file of integers forward, from start to finish

```
/** method that allows us to read a random file from start to finish */
public static void readRAFForward(String fileName){
    //open a random access file in read mode only
    try(RandomAccessFile raf = new RandomAccessFile(fileName, "r")){
        /* get the number of integers using the size of the file in
         * bytes divided by 4 to get the number of integers in the file;
         * remember: there are 4 bytes in an integer (or 32 bits)*/
        long numOfInts = raf.length()/4;
        for(long k = 0; k < numOfInts; k++){
            int number = raf.readInt();
            System.out.printf("%d ", number);
        }
    }catch(IOException e){
        System.out.println("Error occurred while reading the file ...");
    }
}//end of readRAFForward
```

Random Access Files – reading a file of integers backwards, from end to start

```
/** method that allows us to read a random file from end to start */
public static void readRAFBackwards(String fileName){
    //open a random access file in read mode only
    try(RandomAccessFile raf = new RandomAccessFile(fileName, "r")){
        /* get the number of integers using the size of the file in
         * bytes divided by 4 to get the number of integers in the file;
         * remember: there are 4 bytes in an integer (or 32 bits)*/
        long numOfInts = raf.length()/4;
        for(long k = numOfInts - 1; k >= 0 ; k--){
            //set the file pointer to the last int   in the file
            raf.seek(k*4);
            int number = raf.readInt();
            System.out.printf("%d ", number);
        }
    }catch(IOException e){
        System.out.println("Error occurred while reading the file ...");
    }
}//end of readRAFBackwards
```

Random Access Files – reading a file forward from a given position

```
/** method that allows us to read a random file forward from a given position */
public static void readRAFForwardFromGivenPosition(String fileName, long position){
    //open a random access file in read mode only
    try(RandomAccessFile raf = new RandomAccessFile(fileName, "r")){
        /* get the number of integers using the size of the file in
         * bytes divided by 4 to get the number of integers in the file;
         * remember: there are 4 bytes in an integer (or 32 bits)*/
        long numOfInts = raf.length()/4;
        //ensure that the position is not smaller than 0, nor greater than numOfInts -1
        if(position < 0 || position > numOfInts - 1){
            System.out.println("Invalid integer position/index ");
            return;//simply return control to the calling method, not a value
        }
        for(long k = position; k < numOfInts; k++){
            //set the file pointer to the given position
            raf.seek(k * 4);
            //use the method readInt to read the integers from the file
            int number = raf.readInt();
            System.out.printf("%d ", number);
        }
    }catch(IOException e){
        System.out.println("Error occurred while reading the file ...");
    }
}//end of readRAFForwardFromGivenPosition
```

Random Access Files – replacing an element at the given position

```
/** method that allows us to replace the integer at the given position */
public static void replaceElementAtGivenPosition(String fileName, long position,
        int newValue){
    //open a random access file in read and write mode
    try(RandomAccessFile raf = new RandomAccessFile(fileName, "rw")){
        /* get the number of integers using the size of the file in
         * bytes divided by 4 to get the number of integers in the file;
         * remember: there are 4 bytes in an integer (or 32 bits)*/
        long numOfInts = raf.length()/4;
        //ensure that position is not smaller than 0, nor greater than file numOfInts -
        if(position < 0 || position > numOfInts - 1){
            System.out.println("Invalid integer position");
            return;//simply return control to the calling method, not a value
        }
        //set the file pointer to the given position
        raf.seek(position * 4);
        //change the value of the int at the given position to the new value
        raf.writeInt(newValue);
    }catch(IOException e){
        e.printStackTrace();
    }
}//end ofreplaceElementAtGivenPosition
```

Random Access Files – writing an entire integer array to a file

```
/** method that allows us to write integers values to a random access file  */
public static void writeRAF(String fileName, int[] integers){
    //open a random access file in read and write mode
    try(RandomAccessFile raf = new RandomAccessFile(fileName, "rw")){
        /* access each element of integers array and write it
         * to file using writeInt() method */
        for(int k = 0; k < integers.length; k++){
            raf.writeInt(integers[k]);
        }
    }catch(IOException e){
        System.out.println("Error occurred while writing to file ...");
    }
}//end of writeRAF
```

Random Access Files – testing the previous methods in the main

```
/** main method to test our methods */
public static void main(String[] args) {
    //first we read a binary file of integers forward
    System.out.println("reading a binary file forward...");
    readRAFForward("ints.dat");
    System.out.println();
    //then we read that file backwards
    System.out.println("reading a binary file backwards...");
    readRAFBackwards("ints.dat");
    System.out.println();
    //then we read it from a given position
    System.out.println("reading a binary file from a given position...");
    readRAFForwardFromGivenPosition("ints.dat", 2); //at index 2 is 25
    System.out.println();
    //then we replace the element at a certain position
    //re-read the file forward again to see if the change took effect
    replaceElementAtGivenPosition("ints.dat", 3, 77);//replace 1 with 77
    System.out.println("reading the binary file after replacing an integer...");
    readRAFForward("ints.dat");
    System.out.println();
    //last, we write a new binary file of integers and read its contents
    int[] array = {1, 2, 3, 4, 5};
    System.out.println("writing an array of integers to a new file...");
    writeRAF("newInts.dat", array);
    readRAFForward("newInts.dat");
    System.out.println();
}//end main
```

Quiz

- What is the difference between sequential and random access?

In sequential access, a file is processed one byte at a time in sequence from start to finish; in random access, we can access arbitrary locations without reading previous bytes.

- 3 modes in which you can open a random access file are: r, w and rw. True or false?

False: there is no w mode.

- An integer has 32 bits, or 4 bytes (since a byte has 8 bits). True or false?

True.

- How many bytes in a double then?

8 bytes in a double (or 64 bits)

- How many bytes in a char?

2 bytes in a char (or 16 bits).