Lab Notes Week 3
Distributed Systems

File Handling and Interface Handling in Java

Note: Optional – revise if not familiar with

Serial access files

- Serial access files
  - Files in which data is stored in physically adjacent locations, often in no particular order, with each new item of data being added to the end of the file.

- Often misnamed sequential files.
  - A sequential file is a serial file in which the data is stored in a particular order, e.g. in account number order. A sequential file is a serial file, but a serial file is not necessarily a sequential file.

Serial access files (I)

- Binary:
  - a compact format determined by the architecture of the particular computers on which the file is to be used.
  - stores data more efficiently (than text).

- Text
  - human-readable format, using ASCII.
  - more convenient for humans.

Text Files

- Require a *java.io.FileReader* object for input and a *java.io.FileWriter* object for output.
- Constructors for these objects are overloaded and may take either of the following arguments:
  - a string (holding the filename)
  - a File object (constructed from the filename).

Constructor examples

- public FileReader(String fileName)
  throws FileNotFoundException

```
1.  FileReader fileIn = new FileReader("accounts.dat");

2.  FileWriter fileOut = new FileWriter("results.dat");


3.  String fileName = "dataFile.txt";
    ---------------------------------
    FileReader input = new FileReader(fileName);
```

Constructor examples (I)

- public FileReader(File file)
  throws FileNotFoundException

- Creates a new *FileReader*, given the *java.io.File* object to read from.

```
//File objects created first…
File inFile = new File(“input.txt”);
File outFile = newFile(“output.txt”);
// Now the FileReader and FileWriter objects…
FileReader in = new FileReader(inFile);
FileWriter out = new FileWriter(outFile);
```

Constructor examples (II)

- If a string lateral is used, the full pathname may be included, but double backslashes are then required in place of single backslashes. E.g.

```
FileWriter outFile = new
    FileWriter(c:\\data\\results.dat”);
```

- A single backslash cannot be used – it would indicated an escape sequence (resenting on of the invisible characters). E.g., tab.
- File names:
  - Can be called anything;
  - Common practice to denote files by a suffix `.txt` (for text) or `.dat` (for data)

java.io.File

- An abstract representation of file and directory pathnames.
-
- Useful for checking whether an input file actually exist. Programs that depend upon the existence of such a file in order to carry out their processing *must* make use of this method.

java.io. BufferedReader and java.io.BufferedWriter

- As FileReader and FileWriter do not provide sufficient functionality or flexibility for reading and writing data from and to files, we need to wrap a BufferedReader around a File Reader object, and a PrintWriter object around a FileWriter object.

```
BufferedReader input =
    new BufferedReader(new FileREader(“in.txt”));

PrintWriter output =
    new PrintWriter(new FileWriter(“out.txt"));
```

After this we can make use of readLine(), print() and println() methods.

readLine() method

- *BufferedReader* method *readLine*():

```
public String readLine()
                throws IOException
```

Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

```
BufferedReader in =
    new BufferedReader(new FileREader("infile.dat"));
String input = in.readLine();
System.out.println("Value read: " + input);
```

print(), println()

- There are several `print`() and `println`() methods in the `PrintWriter` class:

```
File out = new File("outFile.dat");
PrintWriter output = new PrintWriter(new FileWriter(out));
out.println("Test output");
```

- Numeric values will automatically be converted into strings when output by `print` and `println`. When reading such data back, the wrapper class Integer, Double and Float (as appropriate) will need to be used with their corresponding *parse* methods (`parseInt, parseDouble` and `parseFloat)` if any arithmetic or formatting is to be performed on the data.

```
BufferedReader in =
     new BufferedReader(new FileREader("infile.dat"));
String input = in.readLine();
int num = Integer.parseInt(input);
```

`close`() method

- When the processing of a file has been completed, the file should be close via the `close`() method, which is a member of both `BufferedReader` and `PrintWriter`.

  E.g.:

  ```
  in.close()
  ```

- Especially important for output files, to insure that the file buffer has been emptied and all data written to the file. Since file output is buffered, it is not until the output buffer is full that data will normally be written to disc. If a program crash occurs, then any data still in the buffer will not have been written to disc.

Appending to files

- If you wish to *append* data to a serial file, use the following constructor:

```
public FileWriter(File file, boolean append)
        throws IOException
```

  which constructs a *FileWriter* object given a File object. If the second argument is true, then bytes will be written to the end of the file rather than the beginning

- If you construct file with this constructor:

```
public FileWriter(File file)
         throws IOException
```

  if the file already existed, its initial contents will have been **overwritten** (which may or may not have been your intention)

Flushing

- There may be a significant delay between consecutive file output operations while awaiting input from the user, it is good programming practice to use method `flush`() to empty the file output buffer.

  e.g.

  ```
  output.flush()
  ```

- java.io.Writer.flush():

  Flush the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination. Then, if that destination is another character or byte stream, flush it. Thus one `flush`() invocation will flush all the buffers in a chain of Writers and OutputStreams.

End of file handling

- Programming languages differ in detecting an end-of-file (EOF) situation:
  - with some, a program will crash if an attempt is made to read beyond the end of a file;
  - with others (Java), you must attempt to read beyond the end of the file on order the EOF to be detected. With Java, we keep reading until the string has a *null* reference.

```
textMark = input.readLine();
    while (textMark != null){
          ...
          textMark = input.readLine();
    }
```

Random access files

- Serial access files disadvantages:
  - not possible to go directly to a specific record (need to physically read past all the preceding records).
  - not possible to modify records within an existing file (the whole file would have to be recreated).

- Random access files, also called direct access files overcome both of these problems, but have other disadvantages:
  - all the logical records in a particular file must be of the same length.
  - a given string field must be of the same length for all records on the file.
  - numeric data is not in human-readable form.

- Random access files speed and flexibility greatly outweighs the above disadvantages.

java.io.RandomAccessFile

- To create a random access file in Java, we create a *RandomAccessFile* object which support both reading and writing to a random access file.

- If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written.

- The methods():
  - *readInt*, *readLong*, *readFloat*, *readDouble*
  - *writeInt*, *writeLong*, *writeFloat*, *writeDouble*.

Redirection

- The standard input stream *System.in* is associated with the keyboard, whereas the standard output stream *System.out* is associated with the VDU.

- Use '<' to specify the new source of input.

- Use '>' to specify the new output destination.

  E.g.

  ```
  java ReadData < payroll.dat
  ```

  Program *ReadData* begins execution as normal. But, wheneverit executes readLine statement, it will now take its input from the next available line of text in file *payroll.dat.*

  ```
  java WriteData > results.dat
  ```

  Program *WriteData* directs the output of any print and printl statement to file *results.dat*.

Redirection (I)

- Both input and output with same program can be redirected:

```
java ProcessData < readings.dat > results.dat
```

All *readLines* of *ProcessData* program will read from file readings.dat, while all *prints* and *println*s will send output to file *results.dat*.

Command line parameters

- The *java* command allows us to supply values in addition to the name of the program to be executed. These values are called **command line parameters** and are values that the program may make use of.

- Such values are received by method *main* as an array of *Strings.* If this argument is called *arg*, then the elements may be referred to as *arg[0], arg[1], arg[2*]….

  (Java program called Copy.class copies the contents of one file to another)

  ```
  java Copy source.dat dest.dat
  ```

Command line parameters (I)

```
public class Copy{
   public static void main(String[] arg) throws IOException
   {
   //First check that 2 file names have been supplied...
    if (arg.length < 2){
    System.out.println( "You must supply TWO file names.");
    System.out.println("Syntax:");
    System.out.println("java Copy <source> <destination>");
              return;
    }
     …
   // The rest of the code
}
}
```

Interfaces and Anonymous Classes

- *Interfaces* are special types of classes that cannot be instantiated into objects.
  - They are provided only for inheritance i.e. in order to use an interface you must extend it.

- An *anonymous* class is a class that implements (effectively extends to form a class) an interface without requiring its own class definition and class name.
- They are used when the programmer needs to implement an interface to create a class that will only ever be used once.

Interface

- An interface is completely abstract i.e. its methods do not have a body, but must be defined by subclasses Interfaces, therefore cannot be instantiated into objects. They are used to define contracts for classes.

```
interface Being {
    public int getAge();
}
```

- You cannot instantiate an interface, e.g.

```
Being b = new Being();
```

- You can instantiate classes which extend an interface.

```
Person p = new Person("Pat", 55);
```

where

```
class Person implements Being { etc…
```

Interface implementation

```
class Person implements Being {
	private String name;
	private int age;

	public Person(String n, int a) {
		name = n;
		age = a;
	}
	public String getName() {
		return name;
	}
	public int getAge() {
		return age;
	}
	public String toString() {
		return "Person (" + name + ", " + age + ")";
	}
}
```

Interface example

```
public class InterfaceSample {
	public InterfaceSample() {
	  // You can instantiate classes which extend an interface
	   Person p = new Person("Pat", 55);
	   // You can also extend an interface on the fly, by creating an anonymous class which overrides // the
	required method. This approach is used when you are extending an interface to create a class
	   // that you will use only once. If you would reuse the class, it should be properly  defined.
		Being b = new Being () {
			public int getAge() {
				return 0;
			}
			public String toString() {
				return "This is just some instance of a daft anonymous class";
			}
		};
		System.out.println(p.getAge());
		System.out.println(b.getAge());
		System.out.println(p);
		System.out.println(b);
	}
	public static void main(String args[]) {
		new InterfaceSample();
	}
}
```