CS3310 – Kaminski - Fall 2012 World Data App 1.0
Asgn 1 Project Specs *the MainDataStorage part for now*

OVERVIEW

Assignment 1 project includes: 1) the development of the overall structure of the World Data App (the user programs, developer-support utility programs, and classes) as a single project, which will be expanded during the semester and 2) the implementation of the actual data storage file and its access. Indexes will be added in future assignments.

The actual data will be stored in a random access file, **MainData**, using a direct address file structure on id as the primary key. It will be a binary file (rather than a plain ASCII text file).

There are **4 programs** in this project: **Setup, UserApp, AutoTesterUtility, ShowFilesUtility** – see WorldDataProject.zip for further details in the top comments. Java people will use this Java project as the starting point for their assignment. C# people should view the .java text files involved and develop their own WorldDataProject following the designated format and structure of programs and classes.

Batch processing (vs. interactive processing) is used to facilitate testing and the capturing of the running of the programs for submission for grading. Input data for Setup comes from RawData file. All user requests for UserApp come from TransData (transaction) file. And all output (to the user and developer) is sent to a single LogSession file.

Object Oriented Programming (OOP) paradigm is used for Setup and UserApp programs, but the two developer utilities, AutoTesterUtility and ShowFilesUtility are just plain traditional Procedural Paradigm (PP) programs.

There are **5 classes** (besides the main programs). Setup program uses **RawData** class (for all RawData file handling). Setup and UserApp programs share classes for **DataStorage** (for all MainData file handling) and **UserInterface** (for all TransData file and all LogSession file handling) Future assignments will also include **CodeIndex** and **NameIndex** as shared classes for those two programs.

NOTES

What's a Program?

A program is a physically separate chunk of code in its own .java (or .cs) file that contains its own main (or Main) method as the execution starting point. It is independently compile-able and independently executable. So Setup, UserApp and ShowFilesUtility can each be run manually by the developer (you) completely on their own They can also be run automatically by the AutoTesterUtility program.

Information hiding with OOP

The 5 class names each describe WHAT the object is and its functionality without specifying HOW the underlying storage or interaction will be implemented. The 2 programs using the 5 classes are not aware of HOW the data is stored/accessed nor how the UI is implemented. The class name, e.g.,DataStorage, does not mention that storage will be an external file (vs. internal table or external database or cloud storage over the web) or that the file will be a binary and random access using direct address. The UserInterface does not mention that it is batch processing with input transaction file and output log file (vs. interactive processing using a windows app, web app or mobile app). Similarly for the CodeIndex and NameIndex (added in asgn 2) – the programs aren't aware of what data structures will be used for storage, whether they'll be internal or external storage, what kind of search will be used. All such details of HOW things actually work are completely hidden within the class.

THE 5 DATA FILES

1) ?RawData.csv serial file - CSV format – text file
   I WILL PROVIDE THIS (MULTIPLE VERSIONS)
2) ?MainData.bin direct address on id – binary file
   PROJECT CREATES THIS (MULTIPLE VERSIONS)
3) LogSession.txt serial file – text file
   PROJECT CREATES THIS (A SINGLE, CUMMULATIVE FILE)
4) ?TransData.txt serial file – text file
   I WILL PROVIDE THIS (MULTIPLE VERSIONS)
5) ?IndexBackup.bin serial file – binary file
   Including the headerRec, codeIndex, and nameIndex
   PROJECT CREATES THIS
   THIS FILE NOT USED IN ASGN 1

NOTE: ? is the fileNamePrefix - specified by either:
a) the AutoTesterUtility - sent in as a parameter when the program is run
OR
b) the default value, if the program is being run manually by the developer.

?RawData FILE(S)

Record Description

`id` - a positive integer from less than 400 [uniquely identifies a country]
[not necessarily contiguous set of number]
`code` - 3 capital letters (ideally) [uniquely identifies a country]
`name` - all characters (may contain spaces or special characters)[uniquely identifies a country]
`continent` - one of:Africa, Antarctica, Asia, Europe, North America, Oceania, South America
`region` - all characters (may contain spaces)
`surfaceArea` - a positive integer
`yearOfIndep` - an integer or NULL (or a negative integer in a few cases)
`population` - a positive integer or 0 (could be a very large integer)
`lifeExpectancy` - a positive float with 1 decimal place or NULL
THE REST OF THE FIELDS IN THE RECORD ARE NOT USED IN THIS PROGRAM

NOTES on the ?RawData File(s)

1) *AllRawData.csv file is from the MySQL website (tutorial). I did a bit of editing to clean it up and simplify the data slightly for use in this project.*
2) *There are different versions of this file. All have the same format, but different records to test different situations.*
3) *The A2ZRawData is a smaller file which makes testing easier.*
4) *The DupRawData contains duplicate id's to test duplicate-key handling.*

NOTES on .csv Files

1) *These are .csv files (**Comma Separated Values**), so have variable-length fields and thus **variable-length records**.*
2) *A .csv file opens in Excel or Notepad, by default, depending on your computer's default option for .csv type files. Double-click the file to use the default program. To use the other software to open it, right-click the file and select Open With... and select the software.*

LogSession FILE

Status messages appear AT THE APPROPRIATE TIMES i.e.,
- file opened messages generate in the line of code just after opening the file
- file closed messages generate in the line of code just before closing the file
- program started messages generate at the top of the program's main
- program ended messages generate at the bottom of the program's main

[The fileNamePrefix (the ?) is filled in appropriately, of course].

```
**** Setup PROGRAM started
**** Setup PROGRAM ended - 26 items processed - 20 OK - 6 DUPLICATES
**** UserApp PROGRAM started
**** UserAPp PROGRAM ended - 19 transactions processed
**** ?RawData FILE opened
**** ?RawData FILE closed
**** ?MainData FILE opened
**** ?MainData FILE closed
**** ?TransData FILE opened
**** ?TransData FiLE closed
**** LogSession FILE opened
**** LogSession FILE closed
```

NOTE: Duplicate id's (during Setup) show ERROR (for the 2nd, 3rd, …, not the 1st one)

```
  ERROR - duplicate id for Germany          (not inserted) - id 3 is France
```

NOTE: The transaction request is echo'd before the data is shown

```
QI 3
  003 FRA France           Europe      Western Eu    551,500  0843    59,225,700 78.8
QI 002
  ERROR - no country with that id
IN . . .
  OK, country inserted
IN . . .
  ERROR - duplicate id for Germany          (not inserted) - id 3 is France
DE 3
  OK, country deleted - France
DE 2
  ERROR - no country with that ID
LI
ID CODE NAME              CONTINENT   REGION        AREA    INDEP    POPULATION L.EXP
001 KEN Kenya             Africa      Eastern Af    580,367  1963    30,080,000 48.0
003 FRA France            Europe      Western Eu    551,500  0843    59,225,700 78.8
006 ZWE Zimbabwe          Africa      Eastern Af    390,757  1980    11,669,000 37.8
. . .
+ + + + + + + + + + THE END OF DATA + + + + + + + + + +
```

ShowFilesUtility's results look like this with the . . . part fully filled in, of course):

```
MAIN DATA STORAGE - N is 26, MaxID is 39
[RRN] ID CODE NAME              CONTINENT   REGION        AREA    INDEP    POPULATION L.EXP
[001] 001 KEN Kenya             Africa      Eastern Af    580,367  1963    30,080,000 48.0
[002] EMPTY
[003] 003 FRA France            Europe      Western Eu    551,500  0843    59,225,700 78.8
[004] EMPTY
[005] EMPTY
[006] 006 ZWE Zimbabwe          Africa      Eastern Af    390,757  1980    11,669,000 37.8
. . .
* * * * * * * * * * THE END OF DATA  * * * * * * * * * * * * * * * * * * * * *
```

?TransData FILE(S)

One transaction per line, starting with 2-char tranCode

`QI 3`	(i.e., query by id)
`LI`	(i.e., list by id)
`IN` (then whole record like RawData file's record)	(i.e., insert)
`DE 003`	(i.e.,delete rec with id 3)

Other tranCodes will be added in future asgn.

Implementation NOTES

- There should be separate methods for handling each type of transaction (QC, LC, IN, DE) with a switch statement controlling the CALLING of the appropriate method.
- ListAllById does **NOT show empty locations, nor RRNs**! ! ! ! Users don't care about such things. Developers DO – so ShowFilesUtility DOES show where empty locations are and RRNs
- **DeleteById is a DUMMY STUB for asgn 1**

########################## ?MainData FILE(S) ######################

This file: 1) uses a **direct address** file structure on id for random access and
2) it's a **binary** data file (rather than a text file).
It has a **header record** on the front containing N and maxId (both short 16-bit integers).

. +

RANDOM ACCESS (relative) files

- *A random access file is implemented using a **relative file** (a logical concept, not a physical concept, with Windows/Linux OS) - i.e., relative to the front of the file, which record is being referred to the 1st one, the 10th one, . . To refer to ("point to") a particular record in the file, the **relative key** or relative record number (**RRN**) is specified – i.e., 1, 2, . . . N.*
- *Traditionally relative files (unlike arrays) **start** their RRNs (EXCLUDING the Header Record) **at 1, not 0**, i.e., 1st, 2nd, 3d, . . .*
- *Languages like Java, C#, C++, C implement (a physical concept) random access referencing by specifying the relative byte number (**RBN**) rather than the RRN, and use a **seek** command (or some variation) to "move" the file position pointer to the correct byte location (i.e., the 1st byte of the desired record location) in the file. RBNs start **at 0, not 1**.*
- *Random access files need a **mapping algorithm** to map some field in the record to an RRN (generally).*

DIRECT ADDRESS file structure

- *Direct address is the simplest mapping algorithm to map key values (a field in each record) to RRNs.* This project uses **id as the primary key** – i.e., the record with id 12 is stored in relative location 12, the record with id 39 is stored at RRN 39. There will never be an id 0, and there is no RRN 0 (per se).
- *Direct Address files need fixed-length record locations, so **fixed-length records** are used, so **fixed-length fields** are used. So **fixed-length "strings"** (implemented as char arrays) are used rather than regular strings aren't used (which are variable-length).*
- Records are each **written to the file directly** and are NOT stored internally (e.g., in an array – **you'll lose a lot of points if you do that)**. A single record would be constructed internally, but then the whole record is written to the file**. [Implementation issue: Since this is a binary file, it's simpler, programming-wise, to write individual fields to the file rather than a single complete record – more on this in class. But with this approach, put all the individual `write`'s in a single method].**
- Because the input file is just a **serial file** *(i.e., the records are not in any particular order with respect to id*), the file is created using **random access** rather than sequential access.
  - This requires that a **`seek`** to the correct location in the file is done before writing a record to the file or before reading a record from the file.
  - A `seek` needs a **byte-offset** value (the RBN) as a parameter, which is the number of bytes beyond the 1st byte in the file (which is byte 0). *(This is a physical concept, not a logical concept with Java/C#/C++/C/…).*

Calculating the byteOffset for random access

- *sizeOfHeaderRec (in bytes) should be calculated once and for all since it won't change throughout the run of the program*
- *sizeOfDataRec (in bytes) should be calculated once and for all since it won't change…*
- **`byteOffset = sizeOfHeaderRec + ((rrn – 1) * sizeOfDataRec)`**

+ +

Duplicate ID Handling

- The first record to arrive with a particular ID is inserted in its home location where it belongs)
- Subsequent records with that ID are displayed on the LogSession file as ERRORS and do NOT appear anywhere in the MainData file
- In order to detect whether a location is EMPTY, the code first has to read a record at the designated RRN, then check if it's a valid record or an unused location (see below). So each insert will need: Calculate byteOffset/seek/read then decide whether to do a seek/write or issue an duplicate error message
- DO NOT USE OTHER ALGORITHMS FOR DUPLICATE-CHECKING (e.g., an internal check-list). It is important that you experience this EXTERNAL file handling issue in a programming context.
- There is never more than a single RawData record or MainData record in memory at once

***Unused record locations in file** are of 2 types which your code must allow for:*

1) *"all 0 bits" (i.e., 0's in numeric fields and '\0's in char fields). The OS should initialize the file space for you (otherwise..). When deleting a record, the program would insert the "all 0 bits" record.*
2) *"past EOF" locations – the "read fails" case because a random access read has read past the allocated file space provided by the OS when the file was opened*

+ +

Implementation NOTES

- QueryById MUST use DirectAddress and NOT **`linear search (else 0 points)`**
- ReadOneRecord is overloaded – one version for sequential read (no RRN specified) and one version for random read (RRN specified)
- Do not do special checking for transId's > maxId – just use readOneRecord and let it naturally determine that it's an empty location
- The input ?RawData file MAY have duplicate IDs, and so must be allowed for

+ +

***A BINARY file** (not an ASCII text file) – i.e.,*

- *numeric fields are stored as int's, short's, double's etc. as specified below*
- *fixed-length "string" fields, are specified as char arrays rather than variable-length C#/Java strings (with their added length fields) or C/C++ strings (with their added null terminators)*
- *no field-separators*
- *no <CR><LF> as record separators*

Record Description

IMPORTANT: Use this exact format since the ShowFilesUtility program (written by someone else) will be reading it expecting this exact format.

- HeaderRecord contains N and maxId – both are short 16-bit integers
- Record description for all records except the HeaderRecord (with fields in this order)
  - `id` – a 16-bit short integer
  - `countryCode` – 3 char's
  - `name` – 17 char's (left-justified and either space-filled or truncated on right)
  - `continent` – 11 char's (left-justified and space-filled or truncated on right)
  - `region` – 10 char's (left-justified and space-filled or truncated on right)
  - `surfaceArea` – a 32-bit integer
  - `yearOfIndep` – a 16-bit short integer
  - `population` – a 64-bit long integer
  - `lifeExp` – a 32-bit float

############ SEQUENTIAL FILE PROCESSING ALGORITHM ############

Setup, UserApp, ListAllById method and ShowFilesUtility all do basic sequential file processing of their respective input files. They all thus use the traditional ***Sequential File Processing Algorithm****:*

```
open file
loop til EOF
{      get a single record (or line)
       call some method to process that record
}
close file
```

Implementation NOTES:

- Just because the human algorithm uses a read/process loop structure doesn't mean that the implementation (in a programming language) uses that – it MAY instead need a process/read (with priming read) loop structure – depending on what "read" method is used and what "EOF-detection" approach is used.
- Since Setup and UserApp programs use the OOP paradigm, they don't actually deal directly with their input files (RawData and TransData, respectively). The programs call methods in the appropriate class (RawData and UserInterface) to do the actual file handling – e.g., a constructor opens the file, Get1CountryData or GetATransaction methods read a record from the file, a special FinishWithData or FinishWithTransactions methods close the file.
- There is never more than a single RawData record or a single MainData record in memory at once.