CS3310 – Kaminski - Spring 2013
Asgn 2 Project Specs

World Data App 1.0
add MainData storage (DA file)
use BINARY files (MainData & Backup)
and other changes

######################### OVERVIEW OF CHANGES #######################

Asgn2 is a modification of asgn 1. Assume that the project still satisfies A1 requirements unless changes are specified in the A2 requirements. These are the main changes/additions:

A. This adds storage of the **MainData** as an external random access file with id as key to allow ListAll and Query transactions based on id. It also still allows ListAll and Query transactions based on name (using NameIndex) to **show the actual data** rather than just id (as A1 did).

B. A2 uses **binary files** (rather than plain ASCII text files) for both MainData file and NameIndexBackup file. NameIndex's save and load methods will thus need changing to accommodate a binary file. And **PrettyPrintUtility** also has to be changed and extended to deal with these **two binary files**.

C. Because there is now a lot more processing needed for each **raw data record**, RawData class must be changed. It will now handle only things pertaining to the RawData **FILE**, per se (i.e., open it in the constructor, close it in FinishUp method, read a single record/line in InputOneCountry method). The **NEW RawDataRecord class** (in a physically separate code file) will handle splitting the record into fields, the actual field storage itself, and the appropriate getters and setters for cleaning/converting the fields, as appropriate.

D. This new class provides ready-built services and storage for use by UserApp when **processing IN transactions** since the restOfTransRecord is exactly the same as a raw data record/line.

E. There will now be **additional transaction** types and additional transaction handling.
    1) New transaction types: LI (ListAllById), QI (QueryById), DI (DeleteById). [DI is a dummy stub, like DN]. LI and QI handlers are in MainData class.
    2) IN transactions will now need to call both InsertCountry (in NameIndex class) and InsertCountry (in MainData class).
    3) LN (ListAllByName) and QN (QueryByName) transaction handlers (in NameIndex class) will have to call InputThisRecord (in MainData class), supplying the id found in the name index.

F. The actual **MainData record** will be **displayed** to the user (in Log file) for LI and QI transactions as well as for LN and QN transactions (via InputThisRecord). PrepareRecForDisplay (in MainData class) nicely formats the fields into a nice string for UI's WriteThis.

G. An **AutoTesterUtility program** will be used to automate testing and the demo.

H. **Name is now a 15-char** string in nameIndex (truncated or space-filled on the right, as needed) since that's how it's stored in the MainData file

######################### RawData RECORDS ############################

RawDataRecord class

This is a NEW class, added for A2. It contains storage for each individual field in the record (NOTE: you only need to specify those needed eventually in the MainData file - see below) It also contains the getters/setters to the necessary cleaning/converting of those fields (e.g., removing excess quote characters, generating id's, truncating/padding some strings to fixed size, converting some strings to integers/floats, etc.). This class provides a public method to `ConvertRecToApropFields` which controls the processing of a single record (line) into the properly-configured individual fields.

SetupProgram can then use the appropriate getters (as parameters) when it calls the 2 `InsertCountry` methods (1st for mainData, 2nd for nameIndex) - after it calls rawData (file)'s `InputOneCountry` method (which itself calls rawDataRecord's `ConvertRecToApropFields,` supplying it with the wholeRecordAsABigString as a parameter).

Similarly, UserApp can call `rawDataRecord.ConvertRecToApropFields` in the IN transaction handler (in the big switch statement) before calling the 2 InsertCountry methods.

NOTES on the RawData RECORDS

1) *.csv files have **variable-length fields** and thus **variable-length records**.*
2) *But, MainData records need to be **fixed-length records** (since a direct address file structure is used) which means **fixed-length fields** will be needed. So conversions will be needed on the fields to accommodate what's needed (in the getter methods).*
3) *.csv files are text files, so all fields (after splitting) are just **strings**, even where the description below describes them as positive integers, floating point numbers, etc.*
4) *NULL [i.e., "missing data"] is a valid value in the database world which SQL can handle. But for our C#/Java programs, when converting ASCII-digit data into numeric fields, also convert NULL's to zero values.*

Fields in RawData Record (the actual good data portion)

`code` - 3 capital letters [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 embedded spaces)
    THIS FIELDS IS NOT USED IN THIS PROJECT
`surfaceArea` - a positive integer
`yearOfIndep` - an positive integer (usually) 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 floating point number with 1 decimal place or NULL
`gnp` - *("Gross National Product")* a positive integer (max 7 digits) or 0
THE REST OF THE FIELDS IN THE RECORD ARE NOT USED IN THIS PROJECT

######################### TransData FILE(S) ############################

One transaction per line, starting with 2-char tranCode:
    QN, LN, DN,   QI, LI, DI,   IN
To request Query/List/Delete by Name or Query/List/Delete by Id
    or Insert (in BOTH MainData and NameIndex)

There are several different TransData files for testing different things in the project. They're named TransData1.txt, TransData2.txt, TransData3.txt.

So, when the testing and running the demo, an AutoTesterUtility (program) will run the other Programs, and pass in the fileNameSuffix (i.e., 1 or 2 or 3) to UserApp so it knows which TransData?.txt file to use. (It passes the fileNameSuffix to the constructor when setting up the ui object).

<u>Implementation NOTES</u>

- *There must be separate methods for handling each different type of transaction (inside of NameIndex or MainData, as appropriate) with a switch statement in UserApp to control the CALLING of the appropriate method.*
- *ListAll's do **NOT show locations** (e.g., RRNs) while PrettyPrintUtility DOES. Users don't care about such things; developers DO.*
- *DeleteById and DeleteByName are DUMMY STUBS, meaning: the methods EXIST and are called (for TranCodes DI and DN) but the bodies of the methods just send a message to Log file saying . . .*

########################## Log FILE ##

<u>Additional Status messages</u> (besides what's in A1)

```
>> opened MainData FILE
>> closed MainData FILE
```

<u>Transaction handling</u> <u>(NEW & CHANGED FROM A1)</u>

[NOTE: alphanumeric (string/char) fields are LEFT-justified, numeric fields are RIGHT-justified]
[NOTE: the data shown below are not value-wise based on what's actually in the RawData or TransData -the examples below are just to show the required formatting
[NOTE: When displaying a data record, QI, QN, LI, LN all use the same data-record format]
[NOTE: For LI and LN, the . . . part is, of course, filled in with the rest of the records]

```
QI 1
  001 XYZ ExtraLong Repub North America 12,345,678 -1234 1,234,567,890 78.8 1,234,567
QI 102
  ERROR, not a valid country id
QN Germany
  025 DEU Germany         Europe           345,678   987    34,567,890 79.0   234,567
QN Kalamazoo
  ERROR, not a valid country name
QN Russian Federation
  ERROR, not a valid country name
QN Russian Federat
  018 RUS Russian Rederat Europe         1,222,333  1610     1,222,333 57.8   990,888
DI 03
  SORRY, DeleteById not yet operational
DN United States
  SORRY, DeleteByName not yet operational
IN . . .
  OK, country inserted in main data storage
  OK, country inserted in name index
LI
  ID CODE NAME----------- CONTINENT---- ------AREA INDEP ---POPULATION L.EX ------GNP
  001 XYZ ExtraLong Repub North America 12,345,678 -1234 1,234,567,890 78.8 1,234,567
  002 ZWE Zimbabwe        Africa            90,757  1910       669,000 37.8    98,765
  . . .
  025 FRA France          Europe           345,678   987    34,567,890 79.0   234,567
@ @ @ @ @ @ @ @ @ THE END @ @ @ @ @ @ @ @
LN
  ID CODE NAME----------- CONTINENT---- ------AREA INDEP ---POPULATION L.EX ------GNP
  013 AUS Australia       Oceania       12,345,678 -1234 1,234,567,890 78.8 1,234,567
  009 BRA Brazil          Africa         1,222,333  1610     1,222,333 57.8   990,888
  . . .
  002 ZWE Zimbabwe        Africa            90,757  1910       669,000 37.8    98,765
@ @ @ @ @ @ @ @ @ THE END @ @ @ @ @ @ @ @
```

<u>PrettyPrintUtility</u><u>'s results look like this</u>*(with the . . . part fully filled in, of course):*:

```
MAIN DATA FILE
N is 25
RRN>ID CODE NAME----------- CONTINENT---- ------AREA INDEP ---POPULATION L.EX ------GNP
001>001 XYZ ExtraLong Repub North America 12,345,678 -1234 1,234,567,890 78.8 1,234,567
002>002 ZWE Zimbabwe        Africa            90,757  1910       669,000 37.8    98,765
. . .
025>025 DEU Germany         Europe           345,678   987    34,567,890 79.0   234,567
@ @ @ @ @ @ @ @ @ END OF FILE @ @ @ @ @ @ @ @

NAME INDEX
N is 25, RootPtr is 000
[SUB]  - - - Name - - - - - - - - -  DRP  LCh  RCh
[000]  China                         001  003  001
[001]  India                         002  007  002
[002]  United States                 003  004  011
. . .
[024]  Germany                       022  -01  -01
@ @ @ @ @ @ @ @ @ END OF FILE @ @ @ @ @ @ @ @
```

########################## NameIndexBackup FILE ##########################

This is now a BINARY file. (More below). This means that:

- rootPtr & n (in the headerRecord)
  and drp & leftChPtr & rightChPtr fields (in regular records) are **short (16-bit) int's**
- name fields are stored as 15-char fixed-length strings
- no record-separators (i.e., <CR><LF>) or field-separators stored in the file

########################## FYI: a BINARY file ##############################

A binary file means "not an ASCII (Unicode) text (e.g., NotePad) file"
(for this asgn, we're adding a bit to the requirements for a binary file)

- *no record separators (i.e., <CR><LF>) or preceding byte-counts-for-records*
- *no field-separators (like a commas or spaces)*
- *numeric fields are stored as int's, short's, float's. double's etc.(as dictated by the specs)*
- *alphanumeric fields are fixed-length fields and stored as strings (and the number of char's specified in the record description do NOT include the extra C#/Java string-length bytes, or the extra C/C++ string-null-terminator byte)*
- *the fixed-length string fields (of the specified sizes) have the actual data left-justified within the fixed size, and then space-fill or truncation on the right, as needed)*

########################## FYI: EXTERNAL Storage Structure ################

External storage (a file) vs. internal structure (e.g., NameIndex)

- *Records are each written to the file **immediately** after they're "built". All records belonging in the file are not FIRST constructed and temporarily stored internally (i.e., the file built in an array – **you'll lose a lot of points if you do that)** then when all are completed, dumped to the file at the end (as you did with the INTERNAL DATA STRUCTURE for NameIndex). A single record must be constructed internally, but then the whole record is immediately 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 actual* `write`*'s in a single WriteRecord method.*

MainData FILE

This file: 0) is an **external** storage structure (vs. internal)
1) uses a **direct address** file structure on id for random access and
2) it's a **binary** data file (not a text file).

All handling of the MainData file (opening/closing/reading/writing) **MUST** be done inside the MainData class!!!

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, 3rd, . . .*
- *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*** *are used (so all name fields are 15 char's, (space-filled or truncated on the right, with the good data portion left-justified).*
- *The input file is just a* ***serial file*** *(i.e., the records are not in any particular order, really). However, because id's are auto-generated by Insert (during SetupProgram and UserApp's call to Insert), the file is, in effect, a* ***key-sequential file*** *(on id). But to allow for future changes to the id-generation (e.g., in asgn 3), the file must be created using* ***random access*** *rather than sequential access.*
  - *This requires that a* ***seek*** *to the correct location in the file is done before ANY writing a record to the file or before ANY reading a record from the file.*
  - *A seek needs a* ***byte-offset*** *value (the RelativeByteNumber) 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*** *(not hard-coded) once and for all since it won't change throughout the run of the program*
- *sizeOfDataRec (in bytes) should be* ***calculated*** *(not hard-coded) once and for all since it won't change…*
- **`byteOffset = sizeOfHeaderRec + ((rrn – 1) * sizeOfDataRec)`**

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). But the random read version just does its extra steps to seek, then calls the sequential read method.*
- *Do not do special checking for transId's > maxId – just use readOneRecord and let it naturally determine that it's an empty location*
- *There is never more than one record in memory at the same time. So there is no need for more internal data storage than a SINGLE record. That memory location can be reused for every read/write.*

Record Description

1st) Header Record: just N as a 16-bit short integer
2nd) the rest of the records - Regular Record Description (with fields in this order)

`id`	– 16-bit short integer
`countryCode`	– 3 char string
`name`	– 15 char string
`continent`	– 13 char string
~~`region`~~	~~DON'T USE THIS FIELD which was in RawData & IN transactions~~
`surfaceArea`	– 32-bit integer
`yearOfIndep`	– 16-bit short integer
`population`	– 64-bit long integer
`lifeExp`	– 32-bit float
`gnp`	– 32-bit integer

AutoTesterUtility PROGRAM

Since different TransData test files will be used during testing and the demo (to be handed in), rather than you (the developer) having to manually run SetupProgram, UserApp and PrettyPrintUtility multiple times (supplying different TransData test file names), the AutoTesterUtility PROGRAM will be the driver program. More on this in class.

This program has already been set up in the starter C#/Java projects (for A1) – but the code in there needs to be tailored to these requirement and the subsequent demo specs.

- The only parameters that AutoTesterUtility program has to send in is the fixNameSuffix for TransData (i.e., '1' or '2' or '3'). And UserApp takes that parameter (args[0]) and passes it on to the ui object (for use in its constructor when opening the appropriate file).
- For now during your testing (until the DemoSpecs are available), AutoTesterUtility program should:
  - delete the Log file
  - run SetupProgram once
  - inside the loop, run UserApp specifying '1' then '2' then '3'
  - [run PrettyPrintUtility as needed]
- Since UserApp's Main/main arguments are strings, AutoTesterUtility has to send in a string for fileNameSuffix.