CS3310 – Kaminski - Fall 2014
Asgn 1 Project Specs

CountriesOfTheWorld App 0.5

############################ PROJECT OVERVIEW ############################

<u>The final finished app *(someday, after the semester's over)*</u>

1) *<u>User mode</u>:* This is a mobile app which provides users the ability to lookup information for countries of the world. Users will be able to select:
   - a specific country based on name, code or id
   - OR all countries which satisfy some condition based on continent, size, population, etc. (e.g., in Europe, or with populations > 10,000,000 or. . .)
   - OR list all countries in order by name, code, id, etc.
2) *<u>Administrative mode</u>* will also be available which allows modification of the data:
   - insert a new country (e.g., Scotland after Sept 18, 2015?)
   - delete a country (e.g., Sudan after it split into North Sudan and South Sudan)
   - update a country's data (e.g., Great Britain, if Scotland leaves).
3) *<u>Developer mode</u>* will also be available (hidden in Settings) to facilitate testing whenever changes are made to the app.

Asgn 1 is stage 1 in the development of the app using incremental development

<u>A1 Functionality (and some limitations)</u>

- See Code Files section (after this section) and the rest of the specs for more details
- Mobile/GUI will be added later
- Program (main) controllers will be used rather than using User-Event-controlled programming
- A1 uses batch processing for user transaction requests & logging of responses to user
- A log file captures status messages (GUI progress-bars for this will be added later)
- Data/file structures (and their algorithms) for the main dataStorage and indexes will be simple-to-program and so likely inefficient, so as to get the project code nicely organized, the interfaces designed and a basic app up and running
  - dataStorage – parallel arrays
- Only a single index will be included at this point, to simplify development , yet allow for getting code modules set up for future development
- Not all data fields will be used – those can easily be added later
- Some service modules will just be dummy stubs for now (e.g., "code header exists and method is callable – but code body just reports that function not yet working") – those will be specified below
- Raw data will already be cleaned up considerably (manually, by ME), although some cleanup will still need to be done in your code (inside RawData class), including:
  - Ignore records which do not start with INSERT INTO
    - Including comments, DB extra lines, "blank" lines (incl at END)
  - Strip off the DB SQL aspects
    - On front of record: INSERT INTO `Country` VALUES (
    - On end of record:);
  - Remove the single quote's
  - Split a line ("record") into fields based on commas (i.e., it's a CSV file) – keeping only those fields that are relevant for A1
  - IMPORTANT NOTE: Cleanup an INDIVIDUAL **<u>RECORD</u>** as it's read/requested. Do NOT cleanup the THE WHOLE **<u>FILE</u>** PRIOR TO PROCESSING, either manually or in RawData class). There must be only a SINGLE PASS through the file as records are being requested by Setup (a controller)

<u>Asgn 1 Programming Style & Structure Requirements</u>:

- OOP paradigm for UserApp & Setup
- TestDriver & PrettyPrintUtility are just plain, short, 1(ish) method procedural program
- modular programming:
  - 8 physically separate code files: 4 programs, 4 shared object-classes
  - The 8 code files are described in the next section
  - All 8 code files are within the SINGLE CountriesOfTheWorld PROJECT
  - any method > 1 page/screen (ish) is further modularized
- self-documenting code:
  - class/method/variable names match these specs
  - method names describe WHAT it does NOT HOW it's implemented
- top-comments for each of the 8 code modules containing:
  - project name, module name, author name, brief description of module
- information hiding:
  - object data is private and is only accessible outside the class with getters/setters
  - public service methods are public, local methods are private
  - implementation details are in the body of methods, not in the header
  - objects used only in one module are declared in that module
- input stream processing "design pattern" is used for raw data processing and for transaction data [see note below] – that is,
  - UserApp program and Setup program are both controller-programs
  - RawData class and UserInterface class provide services, when requested by these programs. But they do NOT "run the show"

############################## 8 CODE FILES ##############################

<u>4 Separate Programs</u>

[a program is a stand-alone, separately executable chunk of code which has its own main method]
[BUT, a program can also be executed by another program – by calling its main method]

1) **TestDriver** – runs the other 3 programs multiple times – DemoSpecs will specify the exact order, the number of times to run each, and parameter values to pass in to those programs for which fileNames to be used for each execution

2) **Setup** – the controller program which builds the dataTable and nameIndex based on data in rawData. [NOTE: any work on any of those 3 objects is done within each of their classes rather than in Setup itself – Setup is just the "boss" which requests services (calls methods) in those 3 classes]. The controller uses the input stream processing "design pattern" for batch processing.

3) **UserApp** – the controller program which processes transaction requests (from the user interface) using dataTable and nameIndex to get the appropriate data to pass to the user interface. [NOTE: any work on any of those 3 objects is done within each of their classes rather than in UserApp itself – UserApp is just the "boss" which requests services (calls methods) in those 3 classes].
The controller uses the input stream processing "design pattern" for batch processing.
The program contains a big "switch" statement (based on TranCode: IN, DN, DI, SN, SI, AN, AI) to call the appropriate handler method in either (or both) DataTable class or NameIndex class.

4) **PrettyPrintUtility** – displays the backup file to the log file. It is not an OOP. It reads/writes to those files directly (as if your CS1110 intern were doing this).

4 Shared Object-Classes

[These instantiable classes store data and provide public services They don't "DO anything" (pretty much) unless one of the controllers (Setup or UserApp) puts in a request (i.e., calls a public method).]

1. **RawData** – only used by Setup
– this class handles EVERYTHING to do with A1RawData.csv file - including opening & closing the file, reading a record when requested, reporting EOF true/false when requested, cleaning up a record after reading one in (see A1Functionality section above), splitting lines into fields, getters for whatever fields are needed

2. **DataTable** – used by both Setup and UserApp
- this class handles EVERYTHING to do with the data storage of the country data, including inserting data in the appropriate location when requested, deleting data as requested, retrieving the specified data when requested, dumping the entire internal table (including header data) to a backup file when requested (including opening/writing to/closing the file) by either Setup or UserApp

3. **NameIndex**– used by both Setup and UserApp
- this class handles EVERYTHING to do with the name index, including inserting data in the appropriate location when requested, deleting data as requested, retrieving the specified data when requested, dumping the internal table to a backup file when requested (including opening/writing to/closing the file) by either Setup or UserApp

4. **UI** – used by UserApp (not Setup, not PrettyPrintUtility)
- this class handles EVERYTHING to do with A1TransData?.txt file and when UserApp uses Log.txt file. **[NOTE: Setup program and PrettyPrintUtility don't use UI class to access Log.txt file – they just deal with the file directly right within the program].** This includes opening/closing the files, reading lines from the transaction file, splitting the lines into the appropriate fields (as needed) writing lines to the log file.

#################### COMMUNICATIONS AMONG MODULES ####################

Setup

- calls public service methods (which includes the constructor, getters, methods) in:
  - RawData
  - DataTable (using RawData getters to pass nice individual data fields in to it)
  - NameIndex (using a RawData getter for name)
    (& the call to DataTable method would have returned DRP value)
- writes status messages directly to the Log file WITHOUT using UI (since this is just an administrative utility program)

UserApp

- calls public service methods (. . .) in
  - UI to deal with transactions
  - UI to deal with Logging of status messages
  - DataTable
  - NameIndex

DataTable

- Needs to be able to write to the Log file – so UserApp passes the UI object in to it at the appropriate time

NameIndex

- Needs to be able to call DataTable's SelectByID (once the appropriate name/DRP entry has been found) – so the appropriate object must be passed in at the appropriate times

TestDriver

- Calls Setup (its main), UserApp (its main), PrettyPrintUtility (its main)

PrettyPrintUtility

- Doesn't call anybody – it's not an OOP program – it's just a plain vanilla procedural program which you could ask your CS1110 intern to write (pretty much)

Backup.txt file – special note

- Used in both DataTable class and in NameIndex class.
- When writing to it (both Setup and UserApp will call both class's FinishUp methods):
  - DataTable's FinishUp opens it in truncate mode (& closes it)
  - NameIndex's FinishUp opens it in append mode (& closes it)
- When reading from it (to load the data back into the 2 internal tables) at the start of UserApp, both DataTable and NameIndex need to read from it – so who opens/closes it? If DataTable's LoadTable method closes it, then NameIndex's LoadTable would have to open the file, but the file position pointer would again be at the start of the file. A possible (less than elegant) solution:
  - UserApp itself has its own method which opens the file,
    then calls DataTable LoadTable (which doesn't open/close the file),
    then calls NameIndex's LoadTable (which doesn't open/close the file),
    then closes the file

############################ INTERNAL TABLES ##############################

dataTable is an internal table (parallel arrays for the required fields) -
organized as a direct address table based on id as key

nameIndex is an internal table (parallel arrays for the KV & DRP)
- organized as a sequential table based on name as key
- where KeyValue (KV) is name
  and DataRecordPtr (DRP) is the location in dataTable
  where that data record is actually stored (= id)

N needs to be stored (a counter keeping track of number of countries inserted)
-during Setup
-for an IN transaction

NOTE: there will always be the same number of entries in the nameIndex as there are records in the dataTable

SelectByName could use binary search since it's a sequential table on name

############################## DATA FILES ################################

These are all ASCII text files, viewable in NotePad (see note on .csv files below).

1. **A1RawData?.csv** – provided on course website
   A1RawData.csv (available now) **A1RawDataSample.csv** (available soon)
2. **A1TransData?.txt** – provided on course website (soon)
   There are 4 versions of the file: where ? is 1,2,3 or 4
3. **Log.txt** – created by project – written to by Seup, UserApp & PrettyPrintUtility
4. **Backup.txt** – created by project –
   (i.e., a copy of the INTERNAL HeaderData, DataTable & NameIndex)
   – created in DataTable class when requested by either Setup or UserApp
   - appended to in NameIndex class when requested by either Setup or UserApp
   - read back into dataTable when requested by UserApp's
   - read back into nameIndex when requested by UserApp's's
   NOTE: UserApp & Setup don't deal with this file directly – they submit requests to public service methods in DataTable and NameIndex which do the actual file-handling (i.e., in the constructor and FinishUp methods in each class)

^ ^

A1RawData?.csv - record description *(see CountryDataDefinition.txt on website)*

NOTE: Char fields are enclosed in single quotes – **code REMOVES THEM**

- ~~Extra characters for SQL-compatibility: INSERT INTO `Country` VALUES (NOT USED IN THIS PROJECT~~
- code - 3 capital letters [uniquely identifies a country]
- id - 1- 3 digits [uniquely identifies a country]
- name - all chars (may contain spaces or special characters) [uniquely identifies a country]
- continent - one of: Africa, Antarctica, Asia, Europe, North America, Oceania, South America
- ~~region – NOT USED IN THIS PROJECT~~
- area - a positive integer
- ~~yearOfIndep – NOT USED IN THIS PROJECT~~
- population - a positive integer or 0 [which could be a very large integer]
- lifeExpectancy - a positive float with 1 decimal place
- ~~Rest of fields – NOT USED IN THIS PROJECT~~
- ~~Extra characters for SQL-compatibility:); - NOT USED IN THIS PROJECT~~
- ~~<CR><LF> [Linux people beware !!!]~~

NOTES about .csv files
- *Comma Separated Values file → variable-length fields → variable-length records*
- *.csv files are viewable in Excel or Notepad or Wordpad (or…) - which one is determined by your computer's default option for .csv type files (which you can change). Double-click the file to use the default program. To use some OTHER software to open it, right-click the file, select Open With... and select either Excel or Notepad or Wordpad.*

^ ^

A1TransData?.txt description

One transaction per line (ends with <CR><LF>), starts with 2-char tranCode, for example:

```
SN United States                                (i.e., Select by name)
SI 44                                           (i.e., Select by id)
AN                                              (i.e., Select All by name)
AI                                              (i.e., Select All by id)
DN Sudan                                        (i.e., Delete by name)
DI 44                                           (i.e., Delete by id)
IN WMU,240,West Mich Uni,Europe,123,4567,88.9   (i.e., Insert)
    (i.e., code,id,name,continent,area, population, lifeExpectancy)
```

^ ^

Log.txt – description (3 kinds of entries)

NOTES:
- *You **MUST** use my **exact format/wording/spacing/alignment** as shown below ! ! ! !*
- *3 kinds of things written to the Log file:*
  - *Status messages*
  - *Transactions (requests echod & responses written)*
  - *Output from PrettyPringUtility*
- *Open file appropriately*
  - *TestDriver opens it in truncate mode at the start*
  - *Setup opens it in append mode*
  - *UI class (used by UserApp) opens it in append mode.*
- *Data shown below are not necessarily accurate based on the actual RawData files. I'm just showing you the appropriate display format.*
- *The . . . portion of the AN and AI transaction responses and the PrettyPrintUtility is filled in, of course.*
- *Transaction responses do NOT show storage location (i.e., SUBSCRIPT), just the data while PrettyPrintUtility DOES show storage location and the data*

- *When Setup calls DataTable.Insert, the Insert method does NOT write anything to the Log file. However, when UserApp calls DataTable.Insert, there SHOULD BE a reassurance message written to the Log file.*
- *Both use the exact same format string (except for the [LOC] column*
- *Use appropriate formatters to display the data so it aligns:*
  - *RIGHT-justify numeric fields with embedded commas as appropriate*
  - *LEFT-justify char fields and truncated/right-padded as follows:*
    code: 3 columns, name: 18 columns, continent: 13 columns

Log - Status messages appear AT THE APPROPRIATE TIMES
NOTE: you MUST place code appropriately, that is:
- FILE OPENED messages generate in the line of code JUST AFTER opening the file (in the constructor)
- FILE CLOSED messages generate in the line of code JUST BEFORE closing the file (in the FinishUp method)
- CODE STARTED messages generate AT THE TOP of the appropriate method
- CODE FINISHED messages generate AT THE BOTTOM of the appropriate method

```
STATUS > Setup started
STATUS > Setup finished - 239 countries processed
STATUS > UserApp started
STATUS > UserApp finished - 14 transactions processed
STATUS > RawData FILE opened (A1RawDataSample.csv)
STATUS > RawData FILE closed
STATUS > TransData FILE opened (A1TransData3.txt)
STATUS > TransData FILE closed
STATUS > Log FILE opened
STATUS > Log FILE closed
```

Log - Transaction processing (transaction request echoed before data is shown)

```
SN China
   CHN 094 China              Asia          9,572,900 1,277,558,000 71.4
SI 12
   CHN 094 China              Asia          9,572,900 1,277,558,000 71.4
SN United States of America
   ERROR, invalid country name
SN United
   ERROR, invalid country name
SI 500
   ERROR, invalid country id
AN
   CDE ID- NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE
   AFG 001 Afganistan         Asia            652,090    22,720,000 45.9
   . . .
   ZWE 204 Zimbabwe           Africa          390,757    11,669,000 37.8
   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
AI
   CDE ID- NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE
   AFG 001 Afganistan         Asia            652,090    22,720,000 45.9
   . . .
   CHN 094 China              Asia          9,572,900 1,277,558,000 71.4
   . . .
   UMI 239 United States Mino Oceania              16             0  0.0
   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
DN Belgium
   [SORRY, Delete By Name module not yet working]
DI Western Michigan
   [SORRY, Delete By Id module not yet working]
IN SCT,240,Scotland,Europe,12345,98765,72.6
   OK, country inserted (in data storage & name index)
```

Log - PrettyPringUtility results look like this:

```
N: 240
DATA STORAGE
LOC/ CDE ID- NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE
001/ AFG 001 Afganistan         Asia            652,090    22,720,000 45.9
. . .
094/ CHN 094 China              Asia          9,572,900 1,277,558,000 71.4
. . .
240/ SCT 240 Scotland           Europe           12,345        98,765 72.6

NAME INDEX
LOC/ NAME-------------- PTR
001/ Afganistan         001
. . .
240/ Zimbabwe           204
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

######################## NOTES on USING nameIndex ########################

NAME COMPARISON ISSUES
- *Ignore case when comparing – so "mExICO" successfully finds "Mexico"*
- *Ignore trailing spaces when comparing – so "France   " from the TransData file matches "France" in the nameIndex table*
- *HOWEVER, leading spaces would cause a NO MATCH situation – so "   France" from the TransData file would NOT mathch "France" in the nameIndex table*
- *Only treat full-matches as successful, so "United" must NOT match "United States"*
- *For C# use CompareOrdinal method rather than Compare or CompareTo so that special characters follow strict ASCII-order. Java's uses ASCII-order by default.*

SelectByName
- Searches the nameIndex based on the name field (using binary search)
- Uses the corresponding DRP to call DataTable's SelectById
  to print the FULL RECORD DATA

SelectALLByName
- Loops through the nameIndex (which is in NAME ORDER)
- Using each DRP to call DataTable's SelectById to print the FULL RECORD DATA

######################## NOTES on INSERTS & DELETES ########################

Setup calls DataTable's Insert, then calls NameIndex's Insert

When UserApp gets an IN transaction request, so similarly it calls BOTH:
DataTable's Insert AND NameIndex's Insert

When UserApp gets a DN transaction request, so it calls NameIndex's DeleteByName

Which would normally then proceed to call DataTable's DeleteById
But since it's a DUMMY STUB, nevermind at this point for A1

When UserApp gets a DI transaction request, so it calls DataTable's DeleteById
Which would normally then proceed to call NameIndex's DeleteByName
But since it's a DUMMY STUB, nevermind at this point for A1

############################ NOTES on OOP ############################

OOP - Information hiding (WHAT vs. HOW)

Class NAMES and PUBLIC METHODS describe WHAT the object is and its functionality to the "outside world" (other parts of the project). The code BODY handles HOW the underlying storage works and HOW interaction will be implemented.

Users (like Setup and UserApp) of the object classes' methods (RawData, UI, DataTable, NameIndex), only know what the object classes' public service method names are (including getters/setters and constructor), but NOT what's inside the methods NOR what the data is. They are NOT at all aware of:

- *WHERE the RawData field values come from (A data file? Interactive users? A database? A bar-code scanner? Scraped off the web? QR code scanner on your iPhone?) nor HOW it was derived (Any transformations? Record-splitting into fields? Field editing after reading from text-boxes? Floats changed to integers? Metric changed to imperial measures? Field-values calculated or read-in from storage?)*
- *HOW the table is stored & accessed (a direct address table? A sorted table? a BST? An ordered list? A hash table?) nor whether it's an internal or external structure, nor whether it's in memory or a file or a database or in the cloud*
- *HOW the user interface is implemented other than*
    - *TransData comes in ONE transaction at a time, which might be from a file, a database, data entered in a textbox in a web app on a tablet, a QR code scanned in, an interactive user typing at the console, etc.*
    - *output is sent to Log which might be a file, or a database or the console or the screen on a mobile device, etc.*

This makes OOP programs easier to change since all code changes are done within a specific class, with no (few) changes are needed to the main CONTROLLER parts of the project code.

OOP – Public vs. Private

What's public? - and thus describes WHAT's going on and what's KNOWNABLE to the "outside world" (i.e., the main programs and procedural class code themselves)?

- *Class names*
- *Public service method names (including getters/setters and constructors) and their parameters*

What's private? - and thus describes HOW things are stored and IMPLEMENTED and is knowable ONLY to other code within this class, but NOT to the "outside world"?

- *The bodies of the public service methods*
- *Private methods used to help modularize your code - their names, parameters, code bodies*
- *data storage within the class (public getters/setters make data storage accessible to the outside world)*
- *the actual FILE handling:*
    - *data file name declarations*
    - *opening the file (in the constructor)*
    - *closing the file (in a public FinishUp method, named as such so the outside world doesn't know there's a file involved)*
    - *the actual reading/writing of records*
    - *the setting and checking the "EOF switch" (called DoneWithInput so the outside world won't know it's actually a file).*

Object declaration:

- *Declare an object as locally as possible – if it's only used in one procedural class, then declare it there and FinishUp with it in there.*
- *If an object is declared in an outer callING module, and a callED method needs to use it, then the object would have to be passed in as a parameter*

Data FILE classes:

- *File is opened in constructor – fileNameSuffix must be passed in as a parameter.*
- *File is closed in FinishUp method since program can't control when a deconstructor method would actually execute.*
- *Classes for input files need to handling reading from the file and EOF-checking:*
    - *InputARecord method (e.g., Input1Country, Input1Trans) with no mention of "read" since that sounds like the object is definitely implemented as a FILE*
    - *a boolean DoneWithInput (and DoneWithTrans) method with no mention of hitEOF since that sounds like the object is definitely implemented as a FILE.*
- *Classes for output files must open a file appropriately for the situation (in the constructor) – in truncate mode or append mode.*
- *Classes for output files used by various methods in various classes need a DisplayThis method, with the caller supplying what needs to be written out (with no mention of "write" since that sounds like the object is definitely implemented as a FILE). This method is overloaded since*
    - *status messages, IN reassurance messages, DN/DI Error messages calls supply a single pre-formatted string,*
    - *while SI/AI calls supply individual fields from which a string is built*
    - *while SN calls SI, supplying the DRP, then SI calls DisplayThis*
    - *while AN calls SI repeatedly, each time with a different DRP, which in turn calls DisplayThis*
- *Actual data is provided to the caller via getters, and not directly from instance variables.*

################# NOTES on INPUT STREAM PROCESSING #################

Setup and UserApp both do basic sequential processing of their respective input streams. The proper approach is to get a SINGLE input data set (each time Input1Country or Input1Trans is called), then handle it completely – then loop to do that again until DoneWithInput. This allows for the input stream to be coming from a file, from a database result set, an interactive user, a series of users, repeated use of a barCode scanner, etc. The basic algorithm:

```
Get stream ready (e.g., file opened in class's constructor for OOP)
Loop til nothing more arriving from stream
           (i.e., a boolean method in the class which indicates "done")
{   1.  call a method in the input stream class to
               INPUT a SINGLE data set (record)
               which READs & splits the record/line into fields
               and does whatever other cleanup is needed
    2   call a method in the output handler class to
               PROCESS that SINGLE data set
               (using public GETTERs from the input class as
                      parameters to supply the appropriate data)
}
FinishUp with stream (file closed in class's FinishUp method, for OOP)
```

Implementation NOTES:

- *Just because the human algorithm uses a "READ/PROCESS" loop structure doesn't mean that the implementation (in a programming language) necessarily uses that structure. It MAY instead need a "PROCESS/READ (with a priming read)" loop structure – depending on which "read" method is used and what "EOF-detection" approach is used in a particular language.*
- *There is never more than a single RawData record in memory at once. So only a single object is needed for storing a RawData record. New records are stored in the same storage space, replacing the prior record since you only ever need 1 record (and its fields) at once. Similarly for TransData..*
- *You could have a separate class for data RECORDS rather than combining all file AND record handling in a single class.*