CS3310 – Kaminski - Spring 2016 World Data App 0.5
A1 Project Specs *just the Data Storage part for now*

################################ OVERVIEW ################################

A1 project creates **CountryData** storage for the countries of the world to store and access various attributes for each country. The data is stored in a file, structured using direct address, with id as the primary key. The core **UserApp program** allows users to very efficiently search for a country's data by specifying its id number, to access all countries in id-order, to insert new data and/or to delete old data from the storage structure.

Direct Address provides optimal random access (O(1) time complexity) of a country's record in the file by id, and near-optimal key-sequential access of all the records in the file in id-order, since records in the file are physically ordered by id (because it's direct address). Other access keys will be added in future assignments to provide more efficient access by those keys. For A1, however, access by country name or country code or any key other than id, requires a very expensive O(N) linear search of the file – and accessing all countries in order by name or code or any other attribute would require an expensive external sort.

Two other programs are included in the project: **Setup program** creates the CountryData file based on data in RawData file, and **PrettyPrint program** displays the CountryData file in physical order (including un-used storage locations).

OOP is used for both Setup and UserApp programs, including 3 separate classes for CountryData class (shared by Setup & UserApp), UI class (shared by Setup & UserApp), and RawData class (used by Setup). PrettyPrint just uses plain **procedural programming** since it's just a utility to aid the developer. Setup's `main` and UserApp's `main` are just short controllers which utilize methods in the 3 OOP classes noted above.

Batch processing is used with these FILES:

• Setup	- IN: RawData, OUT: Log, OUT: CountryData
• UserApp	- IN: TransData, IN/OUT: CountryData, OUT: Log
• PrettyPrint	- IN: CountryData, OUT: Log

########################## REQUIRED COMPONENTS ###########################

6 physically separate code files (with these names):

3 physically separate programs:	Setup, UserApp, PrettyPrint
3 physically separate OOP classes:	RawData, CountryData, UI

4 data files (with these names):

RawData.csv	- handled entirely by RawData class (for Setup)
CountryData.csv,	- handled entirely by CountryData class (for Setup and UserApp)
Log.txt	- handled entirely by UI class (for Setup and UserApp)
TransData.csv	- handled entirely by UI class (for UserApp)

NOTE: UI class handles TWO files, when used by UserApp, but only ONE file is involved when used by Setup. So UI class's constructor and "destructor" (i.e., finishUp method) will need to know "who called" to know whether to open/close ONE file or TWO files.

NOTE: As a utility (using plain procedural programming) PrettyPrint just uses CountryData.txt file and Log.txt file directly and NOT via the above classes.

################### WHAT CODE MODULES DO (& DON'T DO) ####################

Setup and UserApp MUST be OOP programs, where their `main` is JUST THE CONTROLLER which:

```
Declares appropriate objects
Loops til input object indicates "done with input stream"
    with actual inputting & outputting handled by their appropriate class
    (passing in the ui object, as needed)
Finishes up with declared objects
```

The OOP classes MUST handle EVERYTHING to do with their actual data file(s) and records - i.e., opening the file, closing the file, reading from the file, writing to the file, detecting EOF in an input file (when Setup and UserApp are run). Setup and UserApp MUST have NO KNOWLEDGE as to HOW RawData, TransData, CountryData and UI are implemented – i.e., that there are files involved. These 2 programs just call the public service methods provided by those classes – e.g., constructors, getters, finishWithObject, inputACountry, inpuATrans, storeACountry, doneWithRawData, doneWithTrans, outputThis, etc. etc. etc.

Setup is the communicator between RawData and CountryData. Those 2 classes DO NOT COMMUNICATE with each other directly. Nor is RawData class aware of what fields or what format that CountryData wants for its fields or records. RawData class provides getters for fields, which Setup can use, which it can then send in to some method in CountryData class

Similarly for UserApp's use of UI class for handling TransData file and CountryData class for handling CountryData file. Those 2 classes DO NOT COMMUNICATE with each other (except for logging – see note below). UserApp is the module when handles ALL COMMUNICATION between the 2 classes.

HOWEVER, since "everyone" needs to "log information" to the Log file, the ui object is passed in for method-calls (where necessary for logging) to public service methods (including constructor) in RawData, CountryData and UI.

########################## INPUT STREAM HANDLING ##########################

RawData file and TransData file are just input streams of records. Both Setup and UserApp process their respective files as input streams – they DO NOT store all their input file's records in memory. Both controllers (their `main`'s) use the common "design pattern" for handling input streams – i.e.,

```
Prepare the stream
Loops til stream done {
    Input a single record
    Completely deal with that single record's data
```

```
        }
        finishUp with the stream
```

Of course the appropriate class's method MUST be called to actually HANDLE:

- prepare the stream – call appropriate constructor
- input a single record - call rawData.inputACountry or ui.inputATrans
- done – call rawData.doneWithRawData or ui.doneWithTrans
- deal with a single record – call countryData.storeACountry or countryData.selectACountry or countryData.selectAllCountries or . . . .
- finishUp with the stream – call appropriate finishUp method

############################## RawData.csv FILE ##############################

Comma Separated Value TEXT file with <CR><LF>'s after each record.

A serial file – not key-sequential by id. So CountryData file will be created physically "randomly" rather than physically "sequentially".

Do NOT store the entire RawData file in memory. There will NEVER be more than ONE RawData record in memory at once. So the RawData class only needs storage space for a SINGLE record and a SINGLE set of fields.

The file is only opened ONCE during this entire project – by the constructor of RawData class. Similarly, it's only closed ONCE. Do NOT open/close the file for every read!

Record Description

id - a positive integer from less than 400 [uniquely identifies a country]
        [not necessarily contiguous set of numbers] [not a key-sequential file]
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 – FIELD NOT USED IN CountryData FILE
size - a positive integer
yearOfIndep - an integer or NULL (or a negative integer in a few cases)
        NOTE: RawData class changes a NULL year to a 0
population - a positive integer or 0 (could be a very large integer)
lifeExpectancy - a positive float with 1 decimal place or NULL
        NOTE: RawData class changes a NULL lifeExp to a 00.0
THE REST OF THE FIELDS IN THE RECORD ARE NOT USED IN THIS PROJECT

################################ Log.txt FILE ################################

A plain TEXT file (with <CR><LF>'s after each record).

CAUTION: It's a SINGLE cumulative file, written to by all 3 programs (via UI class methods for Setup and UserApp programs, though written directly to by PrettyPrint). Note that when Setup runs, it should make sure UI's constructor opens it in TRUNCATE mode – the other 2 should make sure the file opens in APPEND mode.

This file is opened ONLY THREE TIMES during this project, once for Setup (by UI's constructor), once for UserApp (by UI's constructor) and once for PrettyPrint (done directly by that program). Similarly, it's only closed 3 times altogether (by UI's finishUp method, for Setup and UserApp). Do NOT open/close the file every time you write to it!

Status messages appear in the Log file AT THE APPROPRIATE TIMES to help the developer (and the grader) to see what happened when. The code must generate these messages JUST AFTER THE EVENT HAPPENED (except for closing the Log file, where you have to generate the message BEFORE you actually close the file). So

- "open file" messages generate in constructors
- "close file" messages generate in finishUp methods
- "program done" messages generate at the end of program's `main`

```
>> Setup done - 84 countries stored in CountryData file
>> UserApp done - 19 transactions processed
>> PrettyPrint done
>> open RawData file
>> close RawData file
>> open CountryData file
>> close CountryData file
>> open TransData file
>> close TransData file
>> open Log file
>> close Log file
```

NOTE: The messages do NOT show up in the Log file in the order shown above! I'm just demonstrating what the message format is!

Transaction handling generates BOTH an echo of the request AND the data/response
[NOTE: LI's . . . part is filled in, of course]

```
LI
ID CODE NAME              CONTINENT        SIZE     YEAR     POPULATION L.EX
001 KEN Kenya             Africa           580,367  1963     30,080,000 48.0
003 FRA France            Europe           551,500  0843     59,225,700 78.8
006 ZWE Zimbabwe          Africa           390,757  1980     11,669,000 37.8
. . .
+ + + + + + + + + + + + + END OF DATA - 26 countries + + + + + + + + + + + +
QI,3
003 FRA France            Europe           551,500  0843     59,225,700 78.8
QI,002
** ERROR: no country with id 2
IN . . .
OK, Germany inserted
DI,3
OK, France deleted
DI,002
** ERROR: no country with id 2
```

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

```
RRN> ID CODE NAME              CONTINENT      SIZE     YEAR    POPULATION L.EX
001> 001 KEN Kenya             Africa         580,367  1963    30,080,000 48.0
002> EMPTY
003> 003 FRA France            Europe         551,500  0843    59,225,700 78.8
```

```
004> EMPTY
005> EMPTY
006> 006 ZWE Zimbabwe          Africa         390,757  1980    11,669,000 37.8
.  .  .
+ + + + + + + + + END OF DATA - 26 records - 5 EMPTY locations + + + + + + + +
```

NOTES:

- *LI transactions, QI transactions and PrettyPrint all use the SAME FORMAT for printing out an actual record – EXCEPT that PrettyPrint preceeds each record with the 3-digit RRN and "> ".*
- *LI transactions ONLY show the GOOD records, while PrettyPrint shows BOTH GOOD records and EMPTY locations.*
- *The fields displayed are NOT what's actually in the data file – they data is FORMATTED (as shown) so that fields ALIGN. Some alpha fields are truncated or have spaces appended to them on the right. Some numeric fields are printed as fixed-width (like id, year and life-expectancy). Some numeric fields have commas inserted as typically displayed (population and surfaceArea) – and are preceded by spaces.*
- *The data shown above is NOT necessarily CORRECT, content-wise – I'm just demonstrating the required FORMAT.*
- *Use the EXACT FORMAT shown above.*
- *When Setup calls countryData.insertACountry – do NOT have the method print a message to Log file. HOWEVER, when UserApp calls countryData.insertACountry – DO have the method print an OK message to Log file.*

############################## TransData.csv FILE ###############################

Comma Separated Value TEXT files <CR><LF>'s after each record.

One transaction per line, starting with a 2-char transCode

QI,3	(i.e., query by id)
LI	(i.e., list all countries by id)
IN, . . . (then 9 csv fields)	(i.e., insert country)
DI, 003	(i.e.,delete country with id 3)

Other transCodes will be added in future asgn's.

NOTES:

- *There should be separate methods for handling each type of transaction (QI, LI, IN, DI) with a SWITCH statement (in UserApp) controlling the CALLING of the appropriate method.*
- *listAllById does **NOT show empty locations, nor RRNs**! Users don't care about such things. Whereas PrettyPrint program **DOES show where empty locations are** AND shows **the RRNs**! Developers DO care about such things.*

############################ CountryData.csv FILE ##############################

CountryData.csv is a RANDOM ACCESS (direct address) file – so we'll need **fixed-length record locations**, even though we won't have fixed-length actual records. Records will be comma-separated values, including after the final field in the record. So we'll need to know (before Setup starts) what the max-length of each field can be. More on this in class.

CAUTION: Setup creates a NEW CountryData file each time it's run (so open in truncate mode – by CountryData's constructor). If Setup is run a 2nd time, the original CountryData file will be over-written. HOWEVER, when UserApp runs, the constructor should open the file in append mode).

This file is a

- .csv file (Comma Separate Values) – commas BETWEEN all fields
- Just a TEXT file with <CR><LF>'s after each record
- Is a FIXED-LENGTH RECORD-LOCATION file
- Contains a series of String fields

NOTE: File may contain ASCII codes (8-bit char's) OR Unicode (16-bit char's).

NOTE: The field sizes below do NOT necessarily match exactly with:
- What's in the RawData file
- What's in the Log file

<u>**Record Description (with String fields in this order, SEPARATED BY COMMAS)**</u>

id – 1 to 3 digits
code – always 3 char's
name – max 17 char's stored (truncate longer names)
continent – 4 to 13 char's (i.e., Asia . . . North America)
NOTE: Region is NOT stored in this file
size – 1 to 8 digits (NOTE: there are NO embedded commas in this field, even though commas appear in the Log file printout)
yearOfIndep – max 5 char's (5 if there's a '-', 4 char's otherwise, or 1 if it originally was a NULL)
population – 1 to 10 digits (NOTE: there are NO embedded commas in this field, even thoughcommas appear in the Log file printout)
lifeExp – allways 4 char's (including the '.' with 1 decimal place after the '.')
padding with SPACES so record fills whole record location
<CR><LF>

<u>**SIZE OF A RECORD LOCATION – number of CHARs**</u>

Max field sizes: 3 + 3 + 17 + 13 + 8 + 5 + 10 + 4	→ 63 char's
<CR><LF> size: 1 + 1	→ 2 char's
Number of commas separators: 8 * 1	→ 8 char's

So total is 73 char's.
That's 73 bytes if you're using ASCII char's
That's 146 bytes if you're using Unicode.

CAUTION: Use the correct number when calculating BYTE OFFSET for SEEKING.