CS3310 – Kaminski - Spring 2013
Asgn 1 Project Specs

World Data App 0.1
the internal NameIndex
(implemented as a BST)

OVERVIEW

Assignment 1 is part of the larger WorldDataApp project (which we'll continue developing in future assignments). Asgn1 implements the NameIndex portion, including:

- creating it (implemented as a Binary Search Tree (BST)) in SetupProgram, and
- searching, viewing and updating it in UserApp program.

A third program, PrettyPrintUtility, displays the backup index file so the developer (and grader) can easily view the file in a nicely-aligned printout. NameIndex is created as an **internal** data structure – so in order to "move it" from one program execution to another (i.e., from SetupProgram to UserApp, and from UserApp today to UserApp tomorrow), it is saved and re-loaded to/from a backup file.

So there are **3 physically separate programs**, all within a **SINGLE** Java/C# project:

1. SetupProgram – creates the internal NameIndex based on data in the RawData file – and saves it to the Backup file
2. UserApp – loads the NameIndex from the Backup file, then processes the transactions in Trans file, sending output to the Log file, then saves the index to the Backup file
3. PrettyPrintUtility – reads/prints the Backup file to the Log file

There are **4 data files** used in the project:

1. RawDataTester.csv (input to SetupProgram – I'll provide this file)
2. NameIndexBackup.txt (output from SetupProgram, input/output for UserApp and input to PrettyPrintUtility)
3. Log.txt (all 3 programs output to this file)
4. TransData.txt (input to UserApp – I'll provide this file)

Batch processing (vs. interactive processing) is used to facilitate testing and the capture of the programs' executions for submission for grading. That is, all 3 programs write to the Log file, and user transaction requests to UserApp come from the TransData file - rather than the console or a Windows or Web form.

Object Oriented Programming (OOP paradigm) must be used for SetupProgram and UserApp programs. But since it's just a quickie developer-utility program, PrettyPrintUtility just uses the traditional Procedural Paradigm (PP) approach.

So there are **3 classes** (besides the 3 main programs):

1. RawData handles all file/record/field handling for the RawData file. It's only used by SetupProgram.
2. NameIndex handles everything to do with the internal name index and its backup file. This class is used by both SetupProgram and UserApp.
3. UserInterface handles everything to do with batch processing including anything to do with the Log file and the TransData file. This class is used by both SetupProgram (for Logging) and UserApp (for getting TransData and for Logging).

NOTE: PrettyPrintUtility writes to the Log file directly
– it does NOT use the UserInterface class to do so.

PROGRAMMING 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 SetupProgram, UserApp and PrettyPrintUtility can each be run individually by the developer (you) completely on their own (by designating the desired program as the start program before clicking the green arrow). The programs must, of course, be run in the correct order – i.e., SetupProgram must be run before UserApp or PrettyPrintUtility. The 3 programs can also be run automatically by another Driver program – which we'll add in a future assignment.*

OOP - Information hiding

The 3 class NAMES and the PUBLIC METHODS each describe WHAT the object is and its functionality to the "outside world", WITHOUT specifying HOW the underlying storage or interaction will be implemented. The code in the 2 actual programs which use the 3 classes are NOT at all aware of:

- *WHERE the RawData field values come from (A data file? Interactive users? A database? A bar-code scanner?) 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 NameIndex data is stored/accessed (a BST? An ordered list? A hash table?) nor whether it's an internal or external structure or even an actual database or the cloud*
- *HOW the UserInterface is implemented (Batch processing? Interactive using the Console? A Windows or Web front end?)*

This makes OOP programs easy to change since all code changes are done within the class, with no changes to the main program code (e.g., batch input/output can be changed to a web app by altering the code in UserInterface – NameIndex could be changed to use an external HashFile rather than an internal BST – RawData could be read from a database rather than a data file).

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 2 programs 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 knowable ONLY to other code within this class, but NOT to the 2 programs)?

- *The bodies of the public service methods;*
- *Private methods (their names, parameters, code bodies);*
- *internal storage within the class, temporary variables,*
- *Data file name declarations, the actual opening/closing of files, the actual reading of files and setting/checking the "done switch".*

RawData FILE

Record Description (for the actual data records, starting with record #2)

Extra characters to make it SQL-compatible: `INSERT INTO `Country` VALUES (`

`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 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 this semester.

Extra characters to make it SQL-compatible: `);`

A <CR><LF> (which Linux people should be aware of)

NOTES

1) *The first record in the data file is a HeaderRecord containing the names of the fields. Your program should ignore this record as it is not actual "data", per se, just "meta-data".*
2) *RawDataTester.csv file is a subset of 25 data records from RawData.csv file (which is all the 239 countries from that time). This is based on data from the MySQL website (tutorial), AllWorldData.csv file, which I modified/edited somewhat to clean it up and simplify it for this project. You will ONLY USE RawDataTester.csv file for your asgn.*
3) *For easier testing, RawDataTester file contains 25 countries with one name starting with A, one with B. . . one with Z (none with X).*
4) *For asgn 1, you only care about the NAME field.*
5) *For asgn 2, you will use the other 7 fields mentioned above.*

NOTES on .csv Files

1) *This is a .csv file (**Comma Separated Values**),*
   and so has variable-length fields
   and so has 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 (which you can change). 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 either Excel or Notepad.*

NOTE on ID

1) *CountryID is NOT a field in the RawData file. This is an auto-generated field (generated within the RawData class) starting with 1 (not 0).*
2) *GetId method returns the id to the caller.*

TransData FILE

One transaction per line, starting with 2-char tranCode

`QN United States`	(i.e., query by name)
`LN`	(i.e., list all by name)
`IN` (then whole RawData csv-type record)	(i.e., insert)
`DN Germany`	(i.e.,delete by name)

Other tranCodes will be added in future asgn:

QI (QueryByID)	DI (DeleteByID)	LI (ListByID)
QC (QueryByCode)	DC (DeleteByCode)	LC (ListByCode).

Implementation NOTES

- *There MUST BE separate methods for handling each type of transaction (QN, LN, IN, DN) with a switch statement controlling the CALLING of the appropriate method. The actual methods are in the NameIndex class (not the actual UserApp program)*
- ***DeleteByName is a DUMMY STUB for now***

Log FILE

Status messages appear AT THE APPROPRIATE TIMES – place code appropriately, that is
- 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

```
>> started SetupProgram
>> ended SetupProgram - 25 data items processed
>> started UserApp
>> ended UserApp - 19 transactions processed
>> started PrettyPrintUtility
>> ended PrettyPrintUtility
>> opened RawDataTester FILE
>> closed RawDataTester FILE
>> opened TransData FILE
>> closed TransData FILE
>> opened Log FILE
>> closed Log FILE
>> opened NameIndexBackup FILE
>> closed NameIndexBackup FILE
```

Transaction processing (transaction request is echo'd before the data is shown)

```
QN Mexico
  Mexico 003 >> 4 nodes visited
QN mExICO
  Mexico 003 >> 4 nodes visited
QN Western Michigan University
  ERROR, not a valid country name >> 6 nodes visited
QN United
  ERROR, not a valid country name >> 5 nodes visited
QN United States of America
  ERROR - not a valid country name >> 5 nodes visited
IN GBR,United Kingdom, . . .
  OK, inserted with ID 26 >> 3 nodes visited
IN DEU,Germany, . . .
  OK, inserted with ID 27 >> 5 nodes visited
QN Germany
  Germany 027 >> 6 nodes visited
DN United States
  SORRY, DeleteByName not yet operational
DN Western Michigan
```

```
  SORRY, DeleteByName not yet operational
LN
  013 Australia
  004 Brazil
  . . .
  014 Zimbabwe
@ @ @ @ @ @ @ @ @ THE END @ @ @ @ @ @ @ @ @
```

PrettyPrintUtility's results look like this:

```
N is 27, MaxID is 27, RootPtr is 000
[SUB]  - - - Name - - - - - - - - - -  DRP  LCh  RCh
[000]  China                           001  003  001
[001]  India                           002  007  002
[002]  United States                   003  004  011
. . .
[026]  Germany                         027  -01  -01
@ @ @ @ @ @ @ @ @ END OF FILE @ @ @ @ @ @ @ @ @
```

######################### NAME INDEX (as a INTERNAL BST) #############

- BST's and their algorithms will be discussed in class. & see readings on course website.
- This is INTERNAL index. That means it MUST BE
  - 1) built entirely IN MEMORY during SetupProgram's run, and
  - 2) stored IN MEMORY all during UserApp's run.
- It is only stored on a file as a way to PORT it from SetupProgram to UserApp program, and from UserApp (run today) to UserApp (run tomorrow) so the data's not LOST when either program stops executing. It is NOT considered as an EXTERNAL index because it is NOT being PROCESSED from the FILE itself.
- This uses **array storage** for node storage. *[More on this in class. This is not the conventional way to store BST's. Nor is this the conventional Array Storage for binary trees as used for heaps. This uses EXPLICIT pointers, not implicit ones. Don't just use what's described in a/the book or on the internet) for "Binary Tree storage using an array" !!!]*
- The internal storage structure uses array storage with explicit "pointers" (i.e., array subscripts) rather than C-style pointers or C# references.
- **Use -1 for "points nowhere". 0 won't work since that's a valid storage location in arrays.**
- A BST data structure includes additional fields besides what's needed for node storage (just as a stack needs a topPtr, a linked list needs a headPtr, . . .).
  - A BST needs a **`rootPtr`**.
  - And because YOU'RE doing the space management for this (because we're using array storage for nodes), we also need **N**. [So, since arrays start with 0, not 1, N indicates the nextEmpty location in the array]. N is a counter which needs initializing and incrementing at the appropriate times.
  - And MaxID also needs to be stored (and saved to the Backup File) since we may someday implement DeleteByName method
- An index NODE looks like:
  - KeyValue (i.e., the Name as a string)
  - DRP (DataRecordPtr) (which FOR NOW is just CountryID)
  - LCh (LeftChildPtr) is the SUBSCRIPT where the left child node is stored
  - RCh (Right ChildPtr) is the SUBSCRIPT where the right child node is stored

NameIndexBackup File

FinishUp method in NameIndex class calls a local method to SAVE the internal index to the external backup file. The constructor used by UserApp calls a local method to LOAD the external backup file data back into the internal index structure in memory. Save and Load have NO IDEA that the internal structure they're dealing with is really a BST – they think it's just an array of size N.

N and RootPtr and MaxID must also be saved as the HeaderRecord in the file.

Notes on Comparisons

- 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
- Only treat full-matches as successful, so "United" should NOT match "United States"
- Use C#'s CompareOrdinal method rather than Compare or CompareTo so that special characters follow strict ASCII-order. Java's uses ASCII-order by default.

Cautions – USE PROPER BST Algorithms

- ***QN Processing*** *The BST is stored in an array, so linear search is possible for QN transactions – but* **DON'T USE LINEAR SEARCH. YOU'LL LOSE LOTS OF POINTS IF YOU DON'T DO A PROPER BST SEARCH.**
- ***LN Processing*** *ListByName* **MUST use InOrderTraversal algorithm. YOU'LL LOSE LOTS OF POINTS IF YOU DO A SORT.**

############ SEQUENTIAL Stream PROCESSING ALGORITHM ############

SetupProgram and UserApp both do basic sequential processing of their respective input stream – with the input file handling all being done inside the RawData and UserInterface classes for the RawData and TransData files, respectively. The basic algorithm:

```
Get stream ready (file opened in class's constructor)
Loop til nothing more arriving from stream
         (i.e., a method or Boolean variable in class indicates "done")
{    1 call a method in the input stream class to get a SINGLE data
          set (record)(which READs & splits the record/line into fields)
      2   call a method in the output handler class to PROCESS that
          SINGLE data set (record)
          (using GETTERs in the input class as parameters)
}
FinishUp with stream (file closed in class's FinishUp method, for OOP)
```

PrettyPrintUtility uses a similar algorithm, but with all work actually being done within the actual program code, since it's not using OOP.

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 structure. 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.*
- *There is never more than a single RawData record or a single MainData record in memory at once.*