<u>What's a Program?</u>

A program is a physically separate chunk of code in its own .java (or .cs) file that contains its own main (or Main for C#) method as the execution starting point for that program. It is independently compile-able and independently executable.

- Each of A1's 3 programs can be run MANUALLY (by you) in the IDE (by setting that particular program as the "starting program" which'll execute when you click the green arrow).
- Setup, UserApp & PrettyPrint can all be contained within a SINGLE NetBeans (or…) (or Visual Studio for C#) project.
- [NOT REQUIRED FOR A1 – just an idea to think about!) Each of the 3 programs could also be run AUTOMATICALLY by some 4th program, say TestDriver. (NOTE: Whenever a human user is tired manually doing things on a computer manually, a useful question to ask: "How can we automate this tedium. . . with a program"). This might be particularly useful if we were doing testing of the programs with MULTIPLE different input files to test various conditions – and thus having to run the 3 programs 20 times. The TestDriver program could call the `main` (or `Main`, for C#) of each of the 3 programs as needed (like from within a loop) sending in various input file names as input String parameters to `main`.

<u>OOP?</u>

Object Oriented Programming paradigm – as opposed to the older more basic programming paradigm, Proc edural Programming (PP). OOP modularizes programs using classes and objects, where both data storage and related methods having to do with a particular object are contained in a separate class from the main program. PP modularizes programs by grouping related "methods" (procedures & functions) together in a separate containers – but data is mainly passed in & out as parameters (or return values). So PP does less "detail-hiding" than OOP does.

You started out in CS1 doing PP (in CS1110 at WMU) - calculations, if's, looping, method-calling, parameter-passing. Then in the 2nd half of CS1 and all of CS2 (CS1110/1120 at WMU) you did all OOP.

<u>Information hiding with OOP</u>

The 3 OOP class names (RawData, CountryData, UI – for UserInterface) (and the objects declared, rawData, countryData, ui) each describe WHAT the object IS without specifying HOW the underlying storage is done nor HOW the interaction/access will be implemented. The 2 OOP programs themselves (Setup and UserApp) ONLY know:

- WHAT SERVICES (public methods (including constructor/getters/setters)) the classes provide (based on their descriptive names, which specify WHAT the method will DO, with NO MENTION of HOW it'll do it),
- and what INPUT PARAMETERS must be supplied, if any (including parameter order and dataTypes),
- and what value is RETURNED, if any (and its datatype).

That is, the "outside world" (outside of an OOP class) ONLY knows the method interfaces, but NOT the method bodies - and certainly NOT the data storage aspects of the class.

The 2 OOP programs do NOT KNOW, for example, that:

- RawData comes from a FILE rather than a database or an interactive user typing in a Windows app's textboxes or a scanner grabbing barcodes or QR codes or some program scraping the web for data or some human who's taken up a web-based microTask and is himself asking Siri for the data or . . .
- RawData is a CSV file rather than a fixed-length-record/field file or whether it's a TEXT file or a BINARY file or . . .
- What RawData's record format is
- UI gets transaction requests from a FILE rather than from an interactive user clicking on radio buttons and entering data in a text box or from a voice requests (which is using voice recognition software) or . . .
- UI output goes to a FILE rather than an iPhone screen nicely formatted or to a Window on a laptop or a database or . . .
- CountryData is stored EXTERNALLY in a FILE rather than an internal table or a database or in the cloud or . . .
- CountryData's file structure is DIRECT ADDRESS rather than a serial or key-sequential or hash file structure - and thus uses RANDOM ACCESS rather than sequential access
- CountryData is a TEXT file rather than a BINARY file

ALL details of HOW things are stored and HOW things work (the data-access algorithms) are buried within the class in its instance variables, code bodies of public methods/constructors and private methods.

<u>.csv Files</u>

- .csv files (**<u>Comma Separated Values</u>**) have variable-length fields, with a comma between contiguous field values. So records are **<u>variable-length records</u>**.
- .csv files a just TEXT files (typically)
- .csv files can be opened in NotePad or WordPad or . . . in Excel (which will use the commas to display each field in a different column. Which software is used by default, when you double-click the file name depends on your computer's default option for .csv type files. You can choose the software used to open the file by doing right-click of the file, then select "Open With…" and then select NotePad or … You can also change the default permanently so .csv files are always opened with NotePad or … Excel.

<u>RANDOM ACCESS (relative) files</u>

- A random access file is implemented using a **relative file**. That is, 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).
- RRN is a "**logical** concept" rather than a "**physical** concept" in modern languages/OS's] – a conceptual idea rather than something that's "really true" an (d "has to be used" in the programming language. Subscripts (or indexes) for arrays, on the other hand, are physical concepts in Java, C#, …

- Traditionally, relative files (unlike arrays) **start** their RRNs (EXCLUDING the Header Record) **at 1, not 0**, i.e., 1st, 2nd, 3rd, . . . (But it's arbitrary since it's only a logical convention. HOWEVER, if the specs for a particular project specify staring at 1, then the program code MUST start at 1. It's only arbitrary for the project designer – the coder MUST follow the specs).
- The physical mechanism for implementing **random access referencing** in languages like Java, C#, C++, C is by specifying the relative BYTE number (the **byteOffset**), rather than the relative RECORD number (RRN). The code then uses 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 so it's ready to do a read or write.
- byteOffset is a physical concept (like array subscripts) which start **at 0, not 1**. So it's NOT arbitrary, as RRN's starting number is.
- Random access files needs TWO **mapping algorithms**:
  1) to map some field in the record (e.g., id for A1) to its RRN
  2) to map the RRN to the byteOffset (which is used by the seek)
- The 2 mappings and the seek must happen before EVERY read and EVERY write to the file when doing **random access**
- NO seek's are needed if **sequential access** (reading/writing the file in **physical** sequence) is being used for a relative file. It would be handled just like a plain data file (e.g., what you did in CS1110).
  - HOWEVER, if the program does BOTH random access and sequential access, then just before starting any sequential access, you'd need ONE seek to move the file position pointer to the start of the file (usually) – e.g., a seek to byteOffset of 0. You can no longer count on having just opened the file (which puts the file position pointer at byte 0 in the file).

LOGICAL ("conceptual" "pretend") vs. PHYSICAL ("really truly") concepts

- A relative files is a logical concept, not a physical concept, with Windows & Linux OS. Physically a file is just a stream of bytes. (Some older OS's and languages specified relative files as a physical type of file and stored this information in the system directory).
- RRN's are logical addresses in Java/C#/C/C++. Thus, it's arbitrary whether one starts with a 1 or 0. HOWEVER, you MUST follow the DESIGNER's decision on this, which should be specified in the project specs. START RRNs with 1, not 0 for A1. (Some languages like COBOL use RRNs as a physical address, and use that mechanism to "seek" to a file location).
- byteOffsets are:
  - physical addresses from an application program's (Java/C#/C/C++) point of view (i.e., relative to CS3310 usage)
  - though they're just logical addresses from a system (OS, disk, CompEng) point of view– with the physical address being some actual physical location on the disc).

Thus you can NOT arbitrarily choose to start with a 0 or 1. Java/C#/C++'s seek assumes the 1st byte in a file is byte 0.

DIRECT ADDRESS structure (for a FILE)

- Direct address (DA) is the simplest random access mapping algorithm to map key values (a field in each record) to RRNs. This project (A1) uses **id as the primary** key on which to structure the data in the file. So DA designates that 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 typically used,
  so **fixed-length fields** are typically used
  so **fixed-length "strings"** (implemented as char arrays) are typically used
  rather than regular strings (which are variable-length).

  HOWEVER, A1Specs say to use fixed-length record locations, but variable-length csv records, where maximum-sizes are specified for each field.
- Records are each **written to the file directly** and are NOT all stored internally as you construct the whole structure. (That is, don't build the entire random access file in a array in memory, then AFTER all input data's stored in that internal structure, dump the internal structure to a file). (**You'll lose a LOT of points if you do that)**. A single record would be constructed internally, and then that whole record is written to the file in the correct location by a writeOneRecord method**.** (This method MAY contain a single write, or multiple write commands, 1 for every field, depending on your language and the classes/methods available in the library).
- 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 EVERY write of a record to the file and BEFORE EVERY read of a record from the file.
  - A seek needs a **byteOffset** value as a parameter, which is the number of bytes beyond the 1st byte in the file (which is byte 0).
- Direct Address is a structure which can be used as an **external storage structure** (a file) or as an **internal storage structure** (an array).
  - It's a very efficient structure for a lookup table, based on the primary key – it gives O(1) time complexity for a single query.
  - It also provides fairly efficient key-sequential access of all the data in the table since it's in physical sequence (except for the extra time spent skipping over the empty locations, if any).

Calculating the byteOffset for Random Access

- IF a file has a Header Record (which A1's CountryData does NOT), then SIZE_OF_HEADER_REC constant (in bytes) should be calculated once and for all (at the top of the class, since it won't change throughout the run of the program
- SIZE_OF_DATA_REC (in bytes) should be calculated once and for all since it won't change throughout the run of the program – and we're of course using fixed-length records since we're using direct address for A1.

- Don't hardcode a number for this, since it is likely to change in the future. Do calculations for sub-parts of the record:
    - SIZE_OF_ALL_FIELDS_MAX = 3 + 3 + 17 + 13 + 8 + 5 + 10 + 4;
    - N_OF_FIELDS = 8;
    - SIZE_OF_ALL_COMMAS_TOGETHER = N_OF_FIELDS – 1;
      (??? Are you putting a comma AFTER the last field OR NOT???)
    - CR_AND_LF = 2;
      (??? Are you a Linux person who's only using LF's, not CR's???)
    - SIZE_OF_ONE_CHAR = 1
      (??? Use 2 if you're using Unicode, 1 if you're using ASCII)
    - SIZE_OF_DATA_REC = . . .
    - SIZE_OF_HEADER_REC = 0;

- **byteOffset = SIZE_OF_HEADER_REC + ((rrn – 1) * SIZE_OF_DATA_REC)**
- NOTE: use rrn rather than id in the calculation, even though they're the same for A1 since they WILL NOT BE THE SAME THING necessarily in future asgn's when we use a different file structure.

<u>**Unused record locations**</u>

2 types might happen, which your code must allow for:

1) "all 0 bits". (NOTE: The OS should initialize the file space for you (otherwise..???)). When deleting a record, the program would insert an "all {" record.
2) "went past EOF" locations – the "read fails" case because a random access read has read past the allocated file space provided by the OS thus far. If you check for that, then you can just write to that location, and the OS will accommodate this larger file. It's the reading-end of things that's problematic.

<u>**Input Stream Processing Algorithm**</u>

- If the input is just a stream of records coming from a file (OR a stream of interactive user-inputs OR a stream of rows from the resultSet of a database query OR a stream of barCodes from Meijer's scanner OR . . .) – then use this "design pattern" ("everybody knows" common convention) for your processing algorithm.
- Setup program's main, UserApp program's main, listAllById method in CountryData class and PrettyPrint program's main all do basic sequential file processing of their respective input files. They all thus use the traditional :ut stream processing algorithm:

```
prepare stream
        (so you've got a connection and/or are at the start of the stream)
loop til done
{       input a single record (or line)
        call some method to process that record
}
finishUp with stream (if needed)
```

- For a FILE that would be:

```
open file
while ! EOF
{       read 1 record
        process the record (send IN appropriate parameters to handler method)
}
close file
```

- Just because the human algorithm uses a read/process loop structure doesn't mean that the implementation (in a programming language) uses that code 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.
- IMPORTANT !!! Setup program and UserApp program's DO NOT THEMSELVES do the actual work described above. For Setup, it's RawData's class that does the opening of the file (in the constructor), closing of the file (in finishUp method), eof-detecting (boolean doneWithInput method or getter), reading a record and splitting it into fields and… (input1Country method). Similarly for UserApp's dealing with TransData file, where all such ACTUAL handling is done in UI class

<u>**Communication patterns among classes for A1(**</u>

Two communication patterns are used among objects:

1. Communicating via the overall controller (i.e., Setup & UserApp program):
    a. rawData and countryData objects do NOT talk to each other directly. All communication between them goes via Setup program as the controller. Setup calls rawData methods, as needed. Setup calls countryData methods, as needed, sending in appropriate parameters.
    b. ui (for handling TransData file) and countryData objects do NOT talk to each other directly. All communication between them goes via UserApp program. UserApp calls methods in UI class in order to obtain a single transaction (and transCode. UserApp calls appropriate countryData method (based on a big switch statement in UserApp), sending in appropriate parameters.
2. Communicating directly with a service object (i.e., Log file handled by ui object). Since every module wants to write to the Log file, including both the main programs as well as rawData object, countryData object, ui object (for TransData file handling), <u>**PASS IN THE ui OBJECT**</u> in various method calls if that method is going to want to log something.

<u>**Other Implementation NOTES**</u>:

- QI transactions MUST use DirectAddress and NOT **linear search (else 0 points)**
- LI transactions MUST use physically sequential processing of CountryData file. **(A sort will result in 0 points).**
- Private method read1Record is overloaded
    - one version for sequential read (no RRN specified)
    - one version for random read (RRN specified) which does its byteOffset calculation and seek, then calls the sequential read's read1Record
- Do not do special checking for transId's > maxId – just use read1Record and let it naturally determine that it's an empty location
- There's never more than 1 RawData record or 1 CountryData record in memory at once.