CSCE 156/156H
Assignment 2 - Project Phase I
Data Representation & EDI

Dr. Chris Bourke

Spring 2020

1 Introduction

Objects represent and model real-world entities. The *state* of an object consists of the pieces of data that conceptually define what that object is. Often it is necessary to be able to transmit objects between different systems, which may use completely different languages and technologies. The transfer of such data is known as Electronic Data Interchange (EDI) and is achieved by translating objects into a platform-independent data format such as XML (Extensible Markup Language) or JSON (JavaScript Object Notation) representations. Once transferred, the data on the other end can then be translated into an object in the second system. Different languages and frameworks have their own terminology for this process (marshaling/unmarshaling, serialization/deserialization, etc.).

The first phase of the project will focus on the design and implementation of several Java classes to model the various entities in the system. For the first phase, your classes may be simple data containers. Full behavior, methods, and inheritance will be required in Phase II and you are encouraged to plan ahead for any modifications that you may need. Your classes will define, conceptually, what each of the entities are (their data and types and accessor/mutator methods) as well as provide means for creating and building instances of those entities.

You will also write a parser to process a collection of "flat" data files containing data on entities in the old system and build instances of each object. These files were dumped from the old system and are in a non-standard semi-colon and comma separated value format.

Finally, you will also implement functionality to serialize your objects into a data interchange format. Specifically, you will serialize them to XML and/or JSON formats.

2 Data Files

The data dumped from the old system is separated into several files. A full example of well-formatted input and acceptable output has been provided. However, you will also be required to develop your own non-trivial test case. In general, you may assume that all data is valid and properly formatted and all data files are named as specified. You should assume that all data files are located in a directory called `data` and output files should be saved to the same directory.

2.1 Persons Data File

Data pertaining to each person on the system is stored in a data file in `data/Persons.dat`. The format is as follows: the first line will contain a single integer indicating the total number of records. Each subsequent line contains semicolon delimited data fields:

- Person Code – the unique alpha-numeric designation from the old system
- Broker data – if the user is not a broker, this field will be empty. Otherwise it will contain two pieces of data separated by a comma. The first piece is either `E` or `J` (indicating an Expert or Junior broker) and the second is the person's SEC identifier
- Name – the person's name in `lastName, firstName` format
- Address – the mailing address of the customer. The format is as follows: `STREET,CITY,STATE,ZIP,COUNTRY`
- Email Address(es) – an (optional) list of email addresses; if there are multiple email addresses, they will be delimited by a comma.

2.2 Asset Data File

Data pertaining to each asset on the system is stored in a data file in `data/Assets.dat`. The format is as follows: the first line will contain a single integer indicating the total number of records. Each subsequent line contains semicolon delimited data fields depending on the type of asset.

- Deposit Accounts have the following format: `code;D;label;apr`
- Stocks have the following format: `code;S;label;quarterlyDividend;baseRateOfReturn;betaMeasure;stockSymbol;sharePrice`

- Private Investments have the following format:
  `code;P;label;quarterlyDividend;baseRateOfReturn;baseOmegaMeasure;totalValue`

All percentage data is in the range $[0, 100]$ so you may need to adjust appropriately. Portfolio data (`data/Portfolios.dat`) will be used in the next phase.

3 Requirements

You are required to design Java classes to model the problem above and hold the appropriate data. The classes you design and implement, their names, and how they relate to each other are design decisions that you must make. However, you should follow good object oriented programming principles. In particular, you should make sure that your classes are designed following best practices.

In addition, your program will load data from the data files, construct (and relate) instances of your objects, and produce the appropriate output files.

It is your choice as to which format you produce, XML or JSON. For XML, your output file names should be `Persons.xml` and `Assets.xml`; for JSON, your output files should be `Persons.json` and `Assets.json` respectively. Your output files should be placed in the same `data` directory as the input files.

There is no need to define a rigorous schema in either case. However, your XML/JSON should be well formatted and valid (in particular, you may need to escape certain characters). You should follow the general structures in the examples provided, though some flexibility is allowed for tag and element names. However, the output should *conceptually* match the expected output. It must also pass any and all validation tests (using the validators listed below). In addition, you may (in fact are *encouraged* to) use a library to convert your Java classes to XML or JSON if you wish. Some common libraries and more information on each of the formats can be found with the following resources:

- XStream, a light-weight XML binding framework (recommended for XML): http://x-stream.github.io/
- Google-gson (recommended for JSON) library to convert between Java objects and JSON: http://code.google.com/p/google-gson/
- W3C's XML Tutorial: http://www.w3schools.com/xml/default.asp
- JSON Introduction: http://json.org/
- An XML Validator: http://www.w3schools.com/xml/xml_validator.asp
- A JSON Validator: http://jsonformatter.curiousconcept.com/

4 Process

For this initial phase, your objects may be simple data containers since the only thing you are doing with them is parsing data files, creating object instances, and exporting them in a different format. However, much of the code you write in this phase will be useful in subsequent phases, so ensure that you have well-designed, robust, bug-free and reusable code.

4.1 Testing

You are expected to thoroughly test and debug your implementation using your own test case(s). To ensure that you do, you are required to design and turn in at least one non-trivial test case to demonstrate your program was tested locally to some degree. Ultimately your application will be tested on multiple test cases through webgrader including those handed in by your peers.

Understand that the webgrader is *not* a sufficient testing tool. Webgrader is a *black-box* tester: you don't have access to its internals, to the test cases, nor can you debug with respect to it.

Also understand what a test case is: it is an input-output that is known to be good using a method independent of your program (calculated or verified by hand). We will use your test case when grading other assignments, so you are encouraged to be adversarial: design test cases to probe and break "bad" code, but stay within the requirements specified above.

There are many tools available that will help you generate test data. Some examples:

- Mockaroo: http://www.mockaroo.com
- Generate Data: http://www.generatedata.com
- JSON Generator: http://www.json-generator.com

5 Artifacts – What you need to hand in

- Your program *must* be runnable from a class named `DataConverter.java` which *must* be in the package `com.tbf`
- Your input/output test case files *must* be in a `data` directory in your project when you hand it in.
- You must follow the instructions for how to build your project as a single JAR file in Appendix B of the Project Description document. Name your JAR file `Project.jar`

- In addition, you will be writing a design document. The first draft of this document is due 1 week prior to this assignment.

5.1 Common Errors

Some common errors that students have made on this and similar assignments in the past that you should avoid:

- Design should come first–be sure to have thought out a design for your objects and how they relate and interact with each other before coding.
- OOP requires more of a bottom-up approach: your objects are your building blocks that you combine to create a program. This is in contrast with a procedural style which is top-down.
- Worry about the design of objects before you worry about how they are created.
- A good litmus test: if you delete your driver class, are your other objects still usable? Is it possible to port them over to some other uses or another application and have them still work without knowledge of your driver program? If yes, then it is probably a good design; if no, then you may need to reconsider what you're doing.
- Objects should be created via a constructor (or some other pattern); an object should not be responsible for parsing data files or connecting to a database to build itself (a Factory pattern is much more appropriate for this kind of functionality).
- Encapsulation should be respected. Appropriate data fields and appropriate types should be defined for each class. Visibility should be restricted with access done through accessor/mutator methods. Any methods or functionality that acts on a class's data should be encapsulated in the class (unless usage of an external library makes it inappropriate). If a value is based on an object's state and that state is mutable, then the value should be recomputed based on the state it was in at the time that the value was asked for.
- Classes should be designed as stand-alone, reusable objects. Design them so that they could be used if the application was changed to read from a database (rather than a file) or used in a larger Graphical User Interface program, or some other completely different framework.