# CSCE 156 – Assignment 5

*Fall 2018*

For details on the business rules and requirements for this project, see the Project Overview document.

## Phase IV – Database Connectivity and Sorted List ADT

You will modify the application you developed in Phases I & II to interact with the database you designed in Phase III. Your application will be modified to persist data into the database and load data from it rather than from local flat data files. Specifically, you will implement an Application Programming Interface (API) to interact with your database using the Java Database Connectivity API (JDBC). Your API will provide methods to load and persist data to your database.

Additionally, you will need to implement a sorted list Abstract Data Type (ADT) for ordering invoices. The reports need to be sorted by invoice total, *highest-to-lowest*. You are not allowed to use any of the standard JDK collections objects or algorithms, nor are you allowed to exploit sorting functionality provided by your MySQL database or to implement a sorting algorithm outside your list class.

For grading purposes, the database you designed in the previous assignment will need to be setup on your MySQL account on the cse server. Your code should be configured to connect and interact with this database. You should add/remove/modify data to your database as needed for your own testing purposes, but when we grade your program, we will clear out all the data using the API described below (that you must implement) and add/test using our own data.

## Primary Goals

- Use JDBC to design an API containing static methods with provided signatures to interact with your database.
- Design and implement an ADT (array based or linked-list based) for invoices with all standard methods.
- Modify the invoice system to read from the database rather than from the flat files.
- Use the ADT to sort the invoices in a particular order(based on invoice total, *highest-to-lowest*) and generate reports (both *summary* and *detailed*) from the database.

  *HINT: Revisit Lab10, Lab11, Lab12 (if needed)*

## Implementation

You will modify the Java classes you developed in the prior phase to make the following updates:

- Your main driver class will retain its functionality. However instead of reading from data files, it will make a connection to your database, load the appropriate data and create the appropriate objects. The main method in this class will output the same summary and detail reports as in the previous phases. It is highly recommended that you implement (and reuse) several

"factory" methods that retrieve instances of your defined classes by loading from your database.

- You have been provided with a new class called `InvoiceData` in which we have defined a collection of methods and their signatures to interact with your database. Please create a package named "`com.ceg.ext`" in your project (if it doesn't exist already) and include the file within the package. You will fully implement each one of these methods, but not change the package name of the class or signature of any of these methods. You may *add* any additional methods that you feel will simplify your task, but you should *not* modify or remove any of the methods already defined. These methods will be called by our grading program to load data from our test cases into your database. Your JAR file will then be called to produce the reports and verify the correct output. In this context, our grading program can be viewed as the "client" program that interacts with your API.

The details of how you implement your sorted list ADT is a design decision that you need to make. Some possibilities may include the following.

- Your list ADT may be array-based, linked-list based, or something else entirely.
- You should implement the standard list ADT methods to facilitate adding, removing, and retrieving/iterating over elements. Keep in mind that *order should be maintained, not imposed or updated/changed by a method call*. You can additionally implement any convenience methods (batch adds, size, get-by-index, etc.) that you feel would simplify interactions with your list ADT.
- You can/should make your implementation *generic* by parameterizing it. You may also consider allowing constructing your list along with a `Comparator` to maintain ordering in your list implementation.
- You may consider making your list implement the `Iterable` interface so that it can be used in an enhanced for-loop.

## Artifacts

You will turn in all of your code as a runnable JAR file using the CSE *webhandin*(`InvoiceReport.jar`) with your source code included in a zip file (`InvoiceReport.zip`). Any library files (such as the mysql connector JAR) must also be included in your JAR file. If you follow the instructions as before, this should be automatic. JAR files can be included in your project like any other external library (instructions are available in *Appendix C* of the *Appendices* document on Canvas). Turn in a hardcopy of the grading rubric with your name(s) and login(s) filled out.

## Design Write-up

You will update and modify your Design Document draft to include details on your new database API and the Sorted List ADT. An updated printed hardcopy will be handed in 1 week prior to the assignment due date. In the new section, make sure to given enough details so that a technically competent individual would be able to reasonably reproduce an implementation of your design. Be sure to address *at least* the following.

- How are records loaded from the database and into Java objects? How does your API persist data to your database?
- What are the various *side-effects* of each method? That is, what other records are consequently removed from the database in order to maintain data integrity?
- What, if any, data validations do you perform? Where does the validation occur? What expectations do you place on the user with respect to the API?
- The API allows a user to submit null data—what consequences does this have on how have you handled it?
- What is/are the public interface(s) to your list ADT class? How is it constructed?
- How is your list class(es) implemented internally? Is it implemented using an array? A linked list? Other?
- How does the list's capacity expand and/or contract in response to the insertion or removal of elements? How is order maintained in your list ADT?
- How are adds/removes taken care of in your list? How do you handle additions to the list?
- Is your implementation general—that is could it hold other types rather than just invoice objects or is it limited?
- Detail your testing strategies and any changes that were made as a result.

## Common Errors, Issues, & Questions

The following is a list of common errors and issues that past students have encountered and lost points on. These are issues that you will need to avoid and/or address.

1. You are not allowed to modify the package name or any of the method signatures or remove any of the methods in the database API (`InvoiceData.java`) provided to you. All of the required methods will be used by our grader program and hence any modifications to the method signatures can cause your program not behaving as desired on the *webgrader*.
2. You are not allowed to use the standard JDK collections library for your sorted list ADT (and your class should not simply be a "wrapper" to any other class). However, you may use the standard JDK collections in other respects or internally to other classes.
3. A sorted list ADT is an ADT that *maintains* an ordering. That ordering should either be defined at instantiation or by design but should *not* be mutable at different points in its lifecycle. It would be wrong to have functionality that allows the outside world to *change* the ordering.
4. Some designs have pre-sorted objects and then given them to the "sorted list" ADT. This is an explicitly bad and incorrect design as it defeats the purpose of having a *sorted list* entirely. The purpose of such an ADT is that the particulars of how it represents and maintains and order is encapsulated in the class and we deal only with the abstract add, remove, and retrieve methods.
5. Some designs have attempted to sort by requiring the user of the class to call an explicit sort-by method for whatever ordering you wanted to impose. This is not an Object-Oriented Design; in OOP we design objects with semantic meaning that should be invariant throughout an object's life cycle (recall the square/rectangle or ellipse/circle problem). An object's definition should not change based on the program state up to that point (this would be a Structural Paradigm,

not OOP). Defining and implementing a list that sorts by the same ordering throughout its life cycle is more in-line with OOP principles.