…Be PROACTIVE and seek help EARLY in the week rather than panicking on the due date.

ALL work on this project is to be done individually. Seeking help is fine but working too closely with other students will result in a violation of the academic policy for all students involved (*Unauthorized Collaboration*). This obviously also includes referencing solutions from ANY external websites (*Unauthorized Aid*). If you use code YOU did not develop, *you must cite your source* in a comment above the line(s) of code indicating it is NOT your own work, otherwise it will be considered *plagiarism* (*representing others' work, whether copyrighted or not, as one's own). Violate KSU Academic Integrity policies at your own risks – sanctions are severe if caught.*

CIS 200: Project 9 (50 points + 10% Extra Credit)
Due SUN, Apr 24th by 11:59pm

Note: Originally stated due date in the syllabus for this project was Friday Apr 22. Decided to give ALL students 2 extra days to complete/submit, but it will **NOT BE ACCEPTED LATE.**

Plagiarism software is being used in this course on all source code submitted. Avoid temptation to violate policy out of desperation – YOU must understand the concepts covered in this project to be successful on the next exam and in subsequent course work.

Assignment Description: Software is often developing in *teams* where each member of the team may be assigned to write a class. For this assignment, you have been given 2 completed classes (*Instructor*, Course) and 1 *almost* completed class (*Textbook*), and you are assigned to write the application class.

From Canvas, download and look over the following provided classes completed by a previous programmer in your department: *Instructor*, *Textbook*, and *Course* (taken from "*Starting Out with Java...*" Gaddis, Tony; 6th Edition; 2015; p.517).

Notice that the *Course* class contains the *number* and *name* of the course *(e.g., CIS 200 Programming Fundamentals)*, as well as info about the *Instructor* of the course (an *object*) and info about the *Textbook* required for the course (also an *object*.) You may modify any of these classes as needed.

Since this is the final *Java* Project, you will (mostly) choose how to design your program (i.e., *what* goes *where*) and the design/content of your input-output (I/O) screen. However, you will be subjectively graded on your interface with the user (with 10%) and your program must contain the following:

1) First, modify the Textbook class by adding an overloaded method called *calcRetailPrice*. The no-arg version of the method will return the retail price that assumes a 25% markup (e.g., if the wholesale price is $20.95 then this method would return the value 26.1875). The 1-arg version has a whole number passed to it that represents the % markup (e.g., *calcRetailPrice* (40) on the same book would return 29.33).

2) Within your **CourseApp** class:

- Create and properly use an ArrayList containing *Course* objects. You will allow the user to enter the information for as many *courses* as they would like. You can choose how to end input. Add each object to the *ArrayList*. Input will include all information needed to create a *Course* object.

  ** No credit will be given for a program that does not utilize an ArrayList of Course objects **

**You may assume that each course is unique (i.e., there will be only ONE CIS 200, ONE CIS 300, etc.)

3) You will decide where to do the following…

a) **Data Validation:** Use Exception Handling to check for and *properly handle* the exceptions below. To properly handle an exception (*if it occurs*), display an error message and allow the user to re-enter input until the exception is no longer thrown.

   You must use exception handling to get points for this portion of the assignment.

   -A character is entered for ANY numeric data in the program

   -An empty string (enter is pressed) for a string, **throw an exception** (do NOT use a loop to validate …*throw* and *catch* an exception)

b) **Once data is validated** and all input has been entered by the user and stored in the *ArrayList*, use the *toString* method on each *course* object in the *ArrayList* to display the required output, allowing the user to press enter to display each, one at a time. (*Modification of existing toString methods may be necessary*.)

-Course *number* and *name* (e.g., CIS 200 Programming Fundamentals)

-*Instructor's Last Name, First Name* (e.g., Doe, John)

-*Instructor's email address* (username with '@ksu.edu' added to the end)

-Required Textbook info (*Title, author, wholesale price*)

-The *retail price* with a given markup (not part of *toString* method)

For retail price, after displaying the above for a given object, ask the user for the % markup (as a whole number) and then call your overloaded *calcRetailPrice* method and display. Allow the user to either enter the % markup (as a whole number) or just press enter for the default markup of 25% (this is the ONE spot you *will* allow just *enter* to be pressed). Remember do not allow the % sign as part of the markup entered (e.g. 45% markup would be entered as 45, not 45%). Display an error msg and have the user re-enter. …*Wholesale* and *Retail* price must be displayed with $ and 2 decimals ($98.95)

c) **Once all output is displayed**, prompt the user to enter a *Course Number (e.g., CIS 200)*

– If found, *delete* that course from your *ArrayList* and indicate that the course has been deleted from the *ArrayList*.

-If not found, display an appropriate error message and allow user to re-enter until a valid *Course Number* is entered. (One option is to create a *HashMap* from the info in the ArrayList for an easy search, but how you achieve this search is up to you. Reminder that course numbers are unique).

d) Lastly, allow the user to enter in one more *Course* and place the course at the START of the *ArrayList*. Again, use your *toString* method to display all current courses in the *ArrayList*. You do NOT need to worry about holding the screen after displaying each nor displaying the *retail price* again on this last set of displays.

Documentation: At the top of ONLY THE COURSEAPP CLASS, add the following comment block

```
/**
* <Full Filename>
* <Student Name / Lab Section Day and Time>
*
* Clearly indicate to the reader of your code what THIS class does (i.e., the purpose of THIS class)
  Detailed enough so outside reader of code can determine the purpose of this class (at least 3-4 sentences)
*/
```

At the top of *EACH* method, excluding *main*, add the following comment block, filling in the needed info:

```
/**
*       (description of the method)
*       @param (describe first parameter)          …if no parameters, don't include
*       @param (describe second parameter)
*       …(list all parameters, one per line)
*       @return (describe what is being returned)  …if nothing is return, don't include  */
```

Requirements

This program will contain FOUR separate class files and ALL must compile (by command-line) with the statement: **javac <filename>.java**

It must then RUN by command-line with the command: **java CourseApp**

****Make sure and test this before submitting, so points are not lost unnecessarily!**

OPTIONAL Extra Credit Challenge: (10% extra credit – 5 points)

If you do the extra credit, you MUST indicate in your initial comments on p.1 of your Proj9.java code *OPTIONAL EXTRA CREDIT INCLUDED*. Please indicate when you submit the assignment if you did the extra credit (#1, #2, or all) so that the TA knows to test your program for these options.

Make the following modifications to ***add*** the functionally to Project 9. You will submit a SINGLE version of this Project to Canvas. Simply add the functionality.

You can do *any* or *all* of the extra credit below.

1) (+3) Use *MVC architecture* by creating a *separate* class to handle the VIEW (Console I/O). Then use the VIEW class to handle ALL I/O within the Project – no print statements or input statements in the Controller class (*CourseApp*). (FYI: Course, Instructor, and Textbook are *model* classes).

2) (+2) Use *MVC architecture* by creating *another* separate class to handle the VIEW (GUI) – this will give you a total of 6 classes. Again, no I/O statements in the Controller class (*CourseApp*). If done properly, *changing the view* should have very minimal effect on the Controller class.

   For testing of #2, include a line in your code that is commented out that creates a *View_GUI* object (instead of a *View_Console* object) that the TA can use to test this portion of your program.

Submission – read these instructions carefully or you may lose points

To submit your project, first create a folder called *Proj9* and COPY ALL FOUR completed files (if doing the extra credit, this will be 5 or 6 files) into that folder. Right-click on that folder and select "*Send To->Compressed (zipped) folder*". This will create file *Proj9.zip*. Only a .zip file will be accepted for this assignment in Canvas.

Only a .zip file will be accepted for this assignment in Canvas. Again, make sure ALL classes compile AND program runs at the command line.

Important: It is the *student's responsibility* to verify that the *correct* file is *properly* submitted. If you don't properly submit the *correct* file, *it will not be accepted after the due date passes*. No exceptions.

Grading: Only what it submitted to Canvas before the deadline will be considered for grading, so submit the CORRECT files. Files can be re-submitted until the deadline but only the LAST submission is graded.

Programs that do not compile/run from the command line will receive a grade of ZERO, regardless of the simplicity or complexity of the error, so make sure you submit the *correct* file that *properly compiles from the command line.* Programs that *do* compile/run will be graded according to the following rubric:

Requirement	Points
**To be considered for grading, *program must include a working ArrayList of Course objects*	
Correct Documentation Heading w/full description on CourseApp class only with proper JavaDoc documentation on EACH METHOD	**2**
Textbook.java (Code) Overloaded *calcRetailPrice* method properly declared	**3**
CourseApp.java (Code)	
Create and properly use an *ArrayList* of *Course* objects, adding each object to the ArrayList	**5**
Properly call the methods in the other classes through the Course objects, including the *calcRetailPrice* method of the Textbook class	**5**
Within Any of the Classes (Code and Execution)	
Allows user to enter as many courses as they desire	**2**
Exception handling added to handle a character entered for ANY numeric value	**5**
Exception handling added to handle an empty string entered for ANY String input (except for retail markup %).	**5**
Properly displays all courses currently in the ArrayList using the *toString* methods of each class. Includes Course *number and name;* Instructor's *Last Name, First Name* and *email address;* textbook info (Title, author, wholesale price) and retail price / Wholesale and Retail Price must be display with $ and 2 decimals ($98.95)	**10**
Properly prompts for courseNumber, searches ArrayList, and deletes if found. If not found, displays msg and prompts for another. Once deleted, displays a msg course was deleted.	**4**
Properly prompts for final course, adds to the START of the ArrayList, and then displays all current courses in the ArrayList (without Retail Price) using the *toString* method.	**4**
(Worth 10%) Subjective evaluation of user interface – Was user interface generally 'user friendly'? Were all input and error prompts clear? Was output clearly labelled so user could clearly identify results?	**5**
Extra Credit: MVC architecture is used with a Console VIEW class created. *No print or input statements in the controller class* (CourseApp)	**+3**
Extra Credit: Additional GUI VIEW class created. *No I/O statements in the controller class* (CourseApp). Line included in code that is commented out to allow testing of this feature.	**+2**
~~**Minus Late Penalty (10% per day)**~~ **(Not accepted late)**	
Total	**50 + 5**