Homework 7

TSquare... Kinda

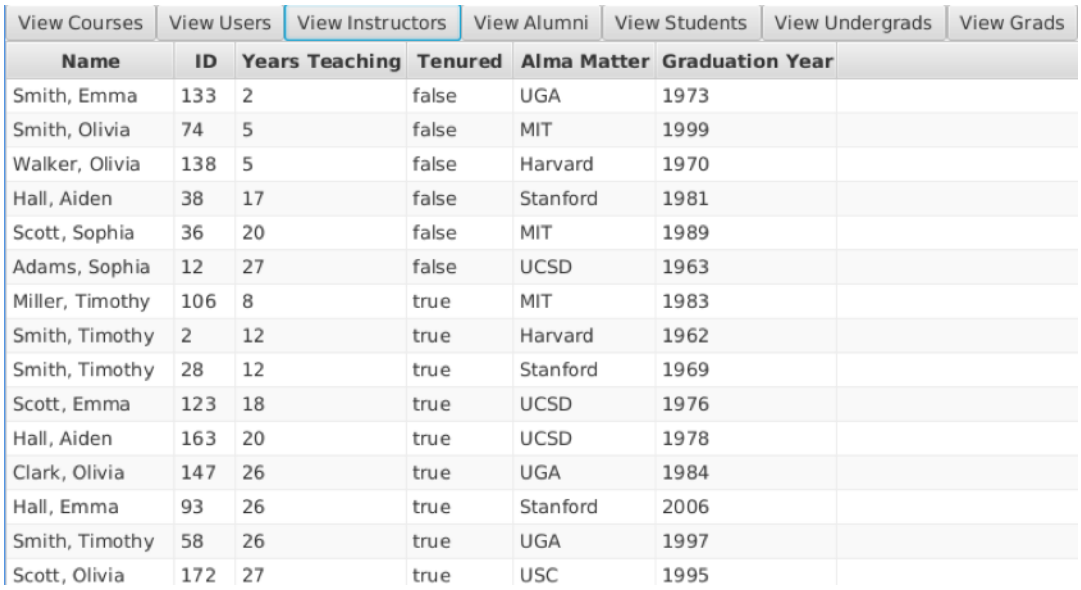
1 Introduction

TSquare is... well it's not terrible, but it could be better. For that reason, in this assignment you will be beginning work on a TSquare replacement, GSquare (ignore Canvas for now). Youll need a solid understanding of polymorphism and interfaces for this homework, so please come to the TA lab if you have questions at any point.

Additionally, be sure to read the **entire document** as there are key details throughout that will help you succeed on this assignment.

2 Problem Description

GSquare is a pretty complex system, including the ability to publish assignments, track grades, host files, etc. Given the week you have for this assignment, youll only be implementing a part of GSquare's functionality: the storing and listing of courses, instructors, and students. A basic GUI and driver class has been provided, so your task will be to make each of the supporting classes representing the main entities within the system. Here's an example of what the completed application will look like (see more in section 5):



Name	ID	Years Teaching	Tenured	Alma Matter	Graduation Year
Smith, Emma	133	2	false	UGA	1973
Smith, Olivia	74	5	false	MIT	1999
Walker, Olivia	138	5	false	Harvard	1970
Hall, Aiden	38	17	false	Stanford	1981
Scott, Sophia	36	20	false	MIT	1989
Adams, Sophia	12	27	false	UCSD	1963
Miller, Timothy	106	8	true	MIT	1983
Smith, Timothy	2	12	true	Harvard	1962
Smith, Timothy	28	12	true	Stanford	1969
Scott, Emma	123	18	true	UCSD	1976
Hall, Aiden	163	20	true	UCSD	1978
Clark, Olivia	147	26	true	UGA	1984
Hall, Emma	93	26	true	Stanford	2006
Smith, Timothy	58	26	true	UGA	1997
Scott, Olivia	172	27	true	USC	1995

3 Provided Files

3.1 GSquare.java

This file serves as the driver and GUI for the application. It functions by randomly generating instances of the classes described in the solution description below, and displaying them them in a sorted list. These lists are sorted by calling each classs compareTo() method, which you will be implementing as described in the next section. You don't need to read through this file to complete the homework, but feel free to take a glance if youre interested. A lot of the concepts used in this file (Javafx, Generics, Collections, etc...) haven't been covered in lecture yet, so dont worry if it doesnt all make sense.

4 Solution Description

4.1 Introduction to Comparable

For our implementation of GSquare, we will be displaying the courses, instructors, and students within our system. When shown, the lists of entities should be sorted according to their "natural ordering." Here, natural ordering refers to the standard way to arrange a list of items. For example, courses might be naturally ordered first by their course code, and in the case of identical course code, by the names of the instructors for the course.

To define the natural ordering of the class, Java provides the Comparable interface, which has a single compareTo method for comparisons. This method is called on an instance of the class, passing in another instance as a parameter: a1.compareTo(a2); Here, a1 and a2 are two instances of a class.

The compare to returns an int that represents the relationship between a2.

- a1.compareTo(a2) returns any negative int: a1 $<$ a2 according to the natural ordering
- a1.compareTo(a2) returns 0: a1 == a2 according to the natural ordering
- a1.compareTo(a2) returns any positive int: a1 $>$ a2 according to the natural ordering

When you implement the compareTo method, it is possible to define the type of the parameter to the compareTo() method. For example, if your class header contains `implements Comparable<Course>`, then you must define your compareTo method as `public int compareTo(Course other) {...}`. This concept is known as "generics," which we will cover in more depth later in the course.

For subclasses that override the compareTo() method, the overridden method must use the same parameter as the superclass. This means that the parameter must be cast to an instance of the subclass in order to compare the classes instance variables:

```
public int compareTo(Superclass other) {
        if (other instanceof Subclass) {
                Subclass casted = (Subclass) other;
                // Compare this to casted
        } else {
                return this.getClass().getName().compareTo(other.getClass().getName());
        }
}
```

Note the else statement in this method, which executes if the parameter is not an instance of the subclass. This is to allow the class to be compared to instances of the parent class. In this case, and as you should do for all of your subclasses that override compareTo(), it compares the name of the classes (.getClass().getName()) and returns the result.

4.2 Classes

Below are the specifications for each of the classes you will be creating. It may look like a lot of work at an initial glance, but dont be too daunted. The implementation required for each file is detailed thoroughly, making it look like more work that it really is. I recommend you work through each file one at a time, starting with the core components (instance data, constructors, getters, etc.) and finishing with the implementation of the compareTo method.

NOTE: For the following classes, you must define each method and variable exactly indicated, otherwise the GSquare.java file will not work properly. Additionally, make sure all getters follow standard formatting (e.g. getCourseCode() for the courseCode variable). If this isnt the case, GSquare.java will throw exceptions during runtime when trying to get data from each class. These exceptions wont actually crash the program, so make sure you check the command line if the application isnt behaving properly!

4.2.1 Course.java

This class represents a course on GSquare.

- Instance Variables:
  - String name (e.g. Intro to Java)
  - int courseCode (e.g. 1331)
  - Instructor instructor
  - Student[] students
- A constructor that takes in a name, courseCode, instructor, and array of students and assigns these parameters correctly to the instance variables.
- Every Course should have the following method:
  - String getInstructorName(): Returns the name of the instructor of the Course.
- Include getters for all instance variables (remember: you must follow standard getter formatting as mentioned above!)
- This class must implement the Comparable<Course> interface, with a matching compareTo(Course other) method.

- Instances of the class have the following natural ordering:
  - Courses with larger courseCodes are ”greater”
  - If the courseCodes are the same, Courses with instructor names later in the alphabet are ”greater” (e.g ”Sophia Miller” > ”Liam Hall”). Note that the String class is comparable.

4.2.2 User.java

Represents a user of the system

- Every User should have the following instance variables:
  - String name: The first and last name of the user (e.g. Stasko, John)
  - int id: The unique identifier for the user. This should be 1 for the first User created, 2 for the second, ..., 100 for the 100th User. Consider using a static variable to keep track of what id to assign to new users.
- Every User should have the following method:
  - String getType(): Returns a string representation of the Class name. You can use this.getClass().getName() to retrieve this string. This is used in GSquare to display the type of each class.
- A constructor that takes in a name and assigns it to the instance variables. This constructor should also assign a unique id to the new User.
- Users are not concrete objects, meaning you should not be able to directly create an instance of one.
- Include getters for all instance variables
- This class must implement the Comparable<User> interface, with a matching compareTo(User other) method.
- Instances of the class have the following natural ordering:
  - Users with larger ids are greater, or equivalent if ids are equal.

4.2.3 Alumnus.java

For GSquare, we want to be able to list all users who had already acquired a degree. As this can include both instructors and certain students, as described below, a good approach is to create an interface to define this requirement. Interfaces serve as a sort of contract, ensuring that the classes that implement the interface have specific methods. This interface ensures that a class provides specific information about graduating, and allows us to create arrays which hold all classes that implement the interface.

- Is an interface
- Methods:
  - String getAlmaMater();
  - int getGradYear();
  - String getName();
  - String getType();

4.2.4 Instructor.java

Represents an instructor in the system.

- Is a subclass of User, and implements the Alumnus interface
- Every Instructor should have the following instance variables:
  - String yearsTeaching: The number of years the instructor has been teaching
  - boolean hasTenure: Whether the instructor is tenured
  - String almaMater: Where the instructor graduated from
  - int gradYear: The year the instructor graduated
- A constructor that takes in yearsTeaching, hasTenure, almaMater, and gradYear and assigns them to the instance variables. This constructor should also take in name to pass to the super constructor.
- Include getters for all instance variables
- This class must override the Comparable<User> interface, with a matching compareTo(User other) method.
- Instances of the class have the following natural ordering:
  - Instructors with tenure are greater
  - In instructors have the same value for hasTenure, they are considered greater if they have been teaching longer.
  - Use Users compareTo if both attributes are the same (hint: super.compareTo)
  - Reminder: See the end of section 4.1 for instructions to handle casting and comparisons to non-Instructor Users.

4.2.5 Student.java

Represents a student in the system.

- Is a subclass of User
- Every Student should have the following instance variables:
  - int creditHours: The number of credit hours the student has taken
  - boolean inState: Whether the student is from in state or not
- A constructor that takes in creditHours and inState and assigns them to the instance variables. This constructor should also take in name to pass to the super constructor.
- Students are not concrete objects, meaning you should not be able to directly create an instance of one.
- Include getters for all instance variables
- This class must override the Comparable<User> interface, with a matching compareTo(User other) method.
- Instances of the class have the following natural ordering:
  - Students with fewer creditHours are ”greater”
  - In students have the same number of credit hours, they are considered ”greater” if they are inState
  - Use Users compareTo if both attributes are the same (hint: super.compareTo)

4.2.6 Year.java

An Enum representing the four possible class standings: FRESHMAN, SOPHOMORE, JUNIOR, SENIOR

4.2.7 Undergrad.java

Represents an undergrad student in the system.

- Is a subclass of Student
- Every Undergrad should have the following instance variables:
  - Year year: The class standing of the student (e.g. Sophomore)
- A constructor that takes in a Year and assigns it to the instance variable. This constructor should also take in name, creditHours, and inState to pass to the super constructor.
- Include getters for all instance variables.
- This class must override the Comparable<User> interface, with a matching compareTo(User other) method.
- Instances of the class have the following natural ordering:
  - Students with a lower year are "greater"
  - Use Students compareTo if the Years are the same

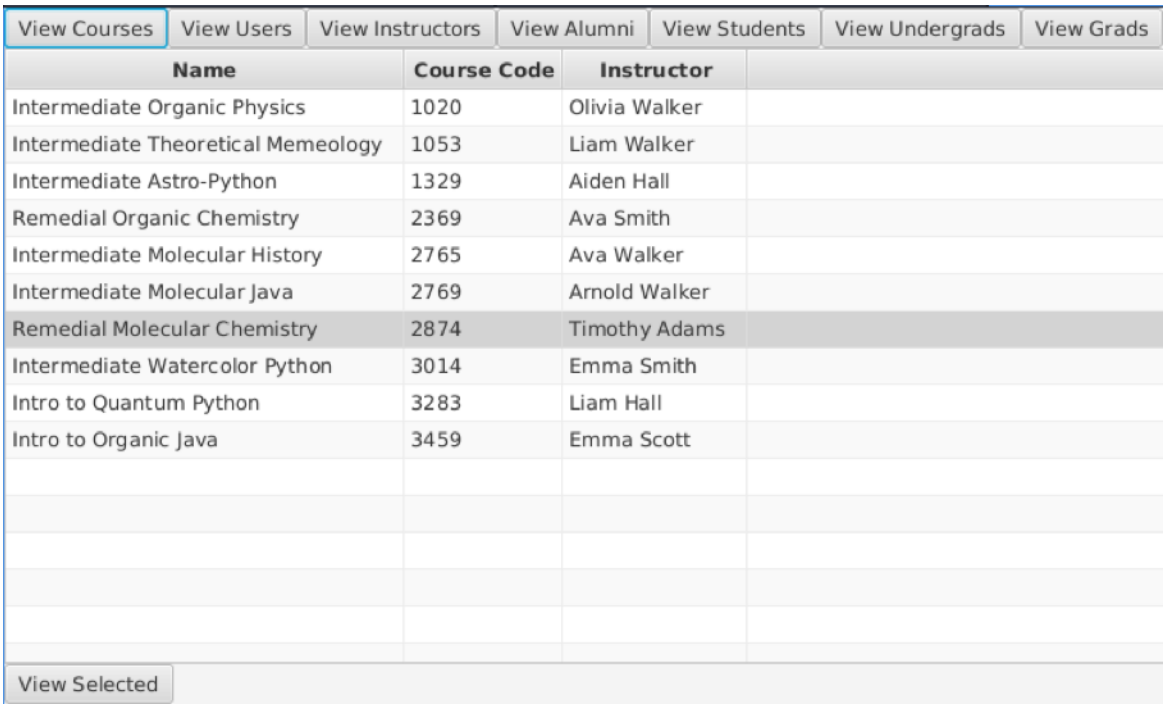
4.2.8 Grad.java

Represents a grad student in the system.

- Is a subclass of Student and implements Alumnus
- Every Grad should have the following instance variables:
  - String almaMater: Where the instructor graduated from
  - int gradYear: The year the instructor graduated
- A constructor that takes in an almaMater and gradYear and assigns them to the instance variables. This constructor should also take in name, creditHours, and inState to pass to the super constructor.
- Include getters for all instance variables
- This class must override the Comparable<User> interface, with a matching compareTo(User other) method.
- Instances of the class have the following natural ordering:
  - Students with a more recent gradYear are "lesser"
  - Use Students compareTo if the gradYears are the same

5 Working Example

The following images show how your implementation should look upon completion. When each class is correctly defined according to the specifications above, the GSquare file should compile and run correctly. The driver (GSquare.java) creates random instances of each class, and stores these in arrays. These arrays are then sorted, using an algorithm which utilizes the compareTo methods of each class. If you have properly implemented your compareTo methods, each list of entities (Grads, Instructors, ...) should be correctly ordered according to their natural ordering. Java by default sorts in ascending order, so the "greatest" instances should be at the bottom of each list.



A sample of a correctly sorted courses list. Note how the course numbers are in ascending order.

View Courses | View Users | View Instructors | View Alumni | View Students | View Undergrads | View Grads

Name	ID	Credit Hours	In State	Type
Smith, Emma	4	35	true	Grad
Miller, Liam	107	127	false	Grad
Adams, Sophia	39	112	false	Grad
Scott, Liam	51	16	false	Grad
Smith, Sophia	3	111	true	Grad
Smith, Arnold	22	36	false	Grad
Clark, Olivia	105	68	true	Grad
Walker, Liam	83	119	false	Grad
Smith, Olivia	14	2	false	Undergrad
Clark, Aiden	98	19	true	Undergrad
Miller, Aiden	50	23	true	Undergrad
Scott, Timothy	28	25	false	Undergrad
Walker, Aiden	5	33	true	Undergrad
Adams, Aiden	96	37	true	Undergrad
Miller, Arnold	43	42	false	Undergrad
Hall, Aiden	21	56	false	Undergrad

A sample of a correctly sorted Students list. Notice how students are first sorted by type, and then by the compareTo methods for each specific class.

View Courses | View Users | View Instructors | View Alumni | View Students | View Undergrads | View Grads

Name	ID	Credit Hours	In State	Alma Matter	Graduation Year
Arnold Scott	74	99	true	USC	2017
Emma Walker	34	55	false	UCSD	2015
Timothy Walker	68	37	false	Harvard	2011
Sophia Clark	71	32	true	USC	2007
Ava Hall	82	117	false	UCSD	2006
Olivia Adams	29	121	true	MIT	2006
Ava Scott	36	127	true	Georgia Tech	2002
Olivia Walker	21	125	true	UGA	1996
Emma Smith	5	4	false	Yale	1993
Aiden Scott	72	114	false	Stanford	1992
Arnold Miller	84	16	false	Georgia Tech	1986
Olivia Adams	93	136	false	USC	1986
Emma Smith	19	76	true	USC	1984
Emma Smith	3	131	true	UCSD	1980

A sample of a correctly sorted Grads list. Note how they are sorted in descending order by graduation year as the natural ordering dictates. Also note how the Grads with the same graduation year are sorted by inState.

6 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

    /**
     * Creates an awesome dog (NOT a dawg!)
     */
    public Dog(){
        ...
    }

    /**
     * This method takes in two ints and returns their sum
     * @param a first number
     * @param b second number
     * @return sum of a and b
     */
    public int add(int a, int b){
        ...
    }
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

6.1 Javadoc and Checkstyle

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

7 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **20** points.
Review the Style Guide and download the Checkstyle jar. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.2.jar -a *.java
Audit done. Errors (potential points off):
0
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **20** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

8 Turn-in Procedure

Non-compiling or missing submissions will receive a zero. NO exceptions

Submit all of the Java source files you modified and resources your program requires to run to T-Square. **Do not submit** any compiled bytecode (`.class` files) or the Checkstyle jar file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code through Checkstyle!

8.1 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

   (a) It helps insure that you turn in the correct files.

   (b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!