P01 Course Enrollment System

Overview

In this assignment, we are going to implement a simple version of a course enrollment system. To ensure a seamless enrollment experience, this application offers the following features and functionalities. For the sake of simplicity, we assume that every course consists of one section.

- **Enrollment:** The system allows a student to enroll in a selected course if (1) they satisfy the pre-requisites of the course and (2) the course did not reach its enrollment capacity. This option also allows a student to enroll from the waitlist into the course, if a seat is secured. We assume that an enrollment permission is automatically issued in this case.
- **Waitlist Option:** If a course has reached its maximum capacity, the system allows a student to join a waitlist, offering them the opportunity to secure a seat if one becomes available.
- **Dropping a course:** The system allows a student to drop a course.
- **Administrative Tools:** The system allows administrators to manage and expand the course capacities (enrollment and/or waitlist capacities), and print course enrollment reports.

Learning Objectives

This assignment allows the cs300 students to:

- **Review** the use of procedure oriented code (prerequisites for this course).
- **Practice** the use of control structures, loops, custom static methods, and arrays in methods.
- **Practice** how to approach an algorithm and decompose a complex problem into sub-problems.
- **Practice** processing text-based user input command lines.
- **Manage** an ordered collection of data (oversize and perfect-size two-dimensional arrays).
- **Develop** unit tests to check the functionality of methods.
- **Familiarize** themselves with the CS300 course style guide requirements and grading tests.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P01 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM CT on Sunday 09/10/2023**. Access to the pre-assignment quiz will be unavailable after its deadline.
+2.5 points	**5% BONUS Points:** Students whose final submissions to Gradescope has a timestamp earlier than 04:59PM CT on Wednesday September 13^{th}will receive an additional 2.5 points toward this assignment's grade on gradescope, up to a maximum total of **45** points.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
25 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.

Assignment Requirements and Notes

(Please read carefully!)

- Pair programming is **NOT ALLOWED** for p01. You MUST complete and submit p01 individually.
- Make sure to read carefully through the WHOLE specification provided in this write-up, and the starter codes before starting the implementation of any method. A sample run is provided in the last section of this specification. Read the instructions TWICE and do not hesitate to ask for clarification on piazza if you find any ambiguity.

External Libraries

- You are **NOT** allowed to ***import*** or ***use*** any **external library** in your `CourseEnrollment` submitted file.
- **The ONLY external libraries** you may use in your submitted files are:
  - `java.util.Arrays`: the `deepEquals`, `toString()`, and `deepToString()` methods in the `CourseEnrollmentTester` class, **ONLY**.

- `java.util.Scanner`, in the `CourseEnrollmentDriver` class **ONLY**

Implementation Requirements

- Only the `CourseEnrollmentTester` and the `CourseEnrollmentDriver` classes contain a **main** method.
- We DO NOT consider erroneous input or exceptional situations in this assignment. We suppose that all the input parameters provided to method calls are valid.
- You are NOT allowed to add any constant or variable in this write-up **outside of any method**.
- You MUST NOT add any **public methods** to your `CourseEnrollment` class other than those defined in this write-up.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define **private static** methods to help implement the different public methods defined in this program, with accordance to this write-up, if needed.
- Any source code provided in this specification may be included verbatim in your program without attribution.
- All String comparisons in this assignment are CASE SENSITIVE.

Test Methods Requirements

- All your test methods should be defined and implemented in your `CourseEnrollmentTester.java`.
- All your test methods must be **public static**. Also, they must take **zero arguments**, **return a boolean**, and must be defined and implemented in your `CourseEnrollmentTester` class.

CS300 Assignment Requirements

This section is VALID for ALL the CS300 assignments

- You are also responsible for maintaining secure back-ups of your progress as you work. The OneDrive and GoogleDrive accounts associated with your UW NetID are often convenient and secure places to store such backups.

- Be sure to submit your code (work in progress) of this assignment on Gradescope both early and often. This will 1) give you time before the deadline to fix any defects that are detected by the tests, 2) provide you with an additional backup of your work, and 3) help track your progress through the implementation of the assignment. These tests are designed to detect and provide feedback about only very specific kinds of defects. **It is your responsibility to implement additional testing to verify that the rest of your code is functioning in accordance with this write-up.**
- You can submit your ***work in progress (incomplete work)*** **multiple times** on gradescope. Your submission may include methods not implemented or with partial implementation or with a default return statement.
- Avoid submitting code which does not compile. Make sure that ALL of your submitted files ALWAYS compile. A submission which contains compile errors won't pass any of the automated tests on gradescope.
- Feel free to **reuse** any of the provided source code in this write-up verbatim in your own submission.
- If starter code files to download are provided, be sure to remove the comments including the `TODO` tags from your last submission to gradescope.
- Your assignment submission must conform to the CS300 Course Style Guide. Please review ALL the commenting, naming, and style requirements.
  - Every submitted file MUST contain a complete file header, with accordance to the CS300 Course Style Guide.
  - All your classes MUST have a javadoc-style class header.
  - All implemented methods including the main method MUST have their own javadoc-style method headers, with accordance to the CS300 Course Style Guide.
- If you need assistance, please check the list of our Resources.
- You MUST adhere to the Academic Conduct Expectations and Advice.

Assignment Development Guideline

Below are steps to help you better approach the cs300 programs. This advice or guideline is not specific to this program. It works for every cs300 programming assignment and we do not plan to include this in the next write-ups. **Read carefully!**

Start early:

We recommend start working on the programming assignment as soon as it is released and no later than Friday. Starting early allows you:

- have time to think about and solve small problems before you get to the hard ones,
- seek the course staff assistance to help you clarify ambiguities and resolve your programming issues, early,
- avoid long assistance waiting time as help requests lines get crazy around the due date of the assignments (Wednesdays and Thursdays),
- complete your program and make your last submission early (by 5pm on Wednesdays) so that you can earn a 5% bonus on a programming assignment grade, and,
- avoid unnecessary stressful situations related to last-minute issues occurring the day of the deadline, and promote your mental health :).

Understand the problem:

Read through the **specification** provided in the **javadocs**! Before you start solving the problem, make sure you have a clear and complete understanding of the problem statement. Identify the inputs, expected outputs, constraints, and any specific requirements of the methods you are going to implement.

Clarify ambiguities:

If the problem specification or the information in the javadocs is unclear or ambiguous, seek clarification from the course staff, immediately. Piazza will be a good place where to first address such issues.

Test methods come FIRST!

Below are general recommendations about testing your program!

- Before writing any code for your methods, it is HIGHLY recommended to implement the tester methods first.
- Write tester methods to test your method with sample of input cases to ensure it works correctly. All that you need to know to write your tester method is the exact signature of the method to test and a clear statement of its inputs, output, expected behavior, and requirements.

- This will allow you to better understand the functionality of each method and define tester scenarios to cover **normal**, **edge**, and **corner** cases. Do NOT forget **edge cases**!
- An edge case is a specific, often extreme or unusual, set of input values or conditions that are at the boundaries or "edges" of the expected or allowable input space for a method or the program. Identifying and handling edge cases is essential for ensuring the robustness and correctness of your program. Here are some examples of edge cases (NOT related to this first assignment).
  - Empty, blank, or null inputs,
  - accessing or considering elements which are at the beginning or at the end of a data structure or arrays,
  - unexpected or incorrect input formats.
  - rare input or exceptional situations.

  A **corner** case involves multiple parameters and input values that require special handling in an algorithm. An edge case involves a unique input parameter at an extreme level.
- A common trap when writing tests is to make the test code as complex or even more complex than the code that it is meant to test. This can lead to there being more bugs and more development time required for testing code, than for the code being tested. To avoid this trap, we aim to make our test code as simple as possible while varying our test scenarios to account edge cases.
- Avoid writing very long tester methods! You can define every unrelated cases in their own tester methods.
- Define **private** helper methods to help you implement every test case on its own, or compare actual and expected results.
- A tester method should return true if and only if NO bug is detected. If it detects any bug it must return false. A tester method is better than another tester method if it can detect more bugs within a broken implementation.

Complete small/simple, easy to code methods first:

Complete small, easy to code tasks first. If you don't know Java syntax, you can learn for a small problem first. Besides, solving many small problems gives you experience and self-confidence to solve larger problems later. A small problem can be directly solved in a few lines of code.

Break down complex problems:

Decompose complex problems into smaller, manageable sub-problems or tasks. This simplifies the problem-solving process and helps you focus on one aspect at a time. This helps also promoting code re-use.

Plan and Pseudocode: Highlight the major algorithmic steps of your method:

Plan your approach to resolve the problem before diving into coding. In the body of your method, write the comments and your algorithm in pseudo-code to sketch out the logical steps you will use. This step helps you think through the algorithm without getting lost in implementation details.

Algorithm design and code implementation:

Design the solution that implements your pseudocode, and translate it into code. You can use branches, conditional statements, loops, data structures, and advanced design approaches, as you see fit. Keep in mind that one problem can have multiple solutions. Try to choose the one that you think best solves the problem. We are going to learn in later chapters throughout this semester how to compare algorithms with respect to their time efficiency.

Test your method, debug and refine:

Test your method using the specific tester method(s) you already developed, considering both normal and edge cases. This will help you detect and locate potential errors and bugs in your implementation. If your method doesn't function as expected, use debugging techniques to identify and fix errors. Step through your code, print intermediate values, and analyze your algorithm's behavior. The Resources page on canvas includes a link to a debugging tutorial video (for eclipse users).

Review the commenting of your code:

Review your code and its commenting and style. Be sure that your last submission follows the directions provided in the CS300 Course Style Guide. Commenting and style represents 10% of p01's grade, for instance.

Practice regularly:

Keep in mind that algorithmic problem-solving (known also as computational thinking) is a skill that develops over time and improves with practice. Work on a variety of algorithmic problems to build your expertise. The Resources page on canvas includes links to websites that many students found helpful to practice coding and problem solving. Be patient and persistent, and don't hesitate to seek help when needed.

Now it is time to start working on the development of this first programming assignment!

1 Getting Started

- To get started, **create** a new **Java17** project within Eclipse. You can name this project whatever you like, but "**p01 Course Enrollment System**" is a good choice. You MUST ensure the following.
  - Make sure that your new project uses **Java 17**, by setting the "Use an execution environment JRE:" drop down setting to "JavaSE-17" within the new Java Project dialog box.
  - DO NOT create any module within your java project. Use the default *src* package.
- **Download** the following files to the ***src*** folder of your project. These files are also available on the P01 Assignment page on canvas.
  1. **CourseEnrollment.java**: Template for the CourseEnrollment class to be implemented. This class contains all the methods implementing the functionalities of this enrollment management application.
  2. **CourseEnrollmentTester.java**: Template for the tester class to be implemented. This class contains all the unit tester methods that you implement to convince yourself with the correctness of your implementation. Note that our autograder (on gradescope) can use your tester methods to test the functioning of good and bad implementations of course staff members.
  3. **CourseEnrollmentDriver.java**: Driver class implementing a text-based command-line interface simulating students enrollments. You will need to complete the implementation of this class as it is directed in a later section of this write-up.
- The **javadocs** of the classes defined in this assignment are accessible through **this link**.

CS300 is a course with students coming with different programming language experiences not necessarily Java. Since this is the first assignment, we provided you with detailed starter codes of the classes to be implemented in this program. This won't be the case for most assignments. You will be required to create the different classes on your own from scratch in the future.

2 Overview of the arrays used in this program

You are going to implement a simple version of a course enrollment management system. Whenever you set up a new course for enrollment, the method will create two separate arrays to store students' records: a ***roster*** for enrolled students and a roster for ***waitlist***ed students.

2.1 Roster to store the records of enrolled students

- **(String[ ][ ] roster, int size)**: An **oversize** two-dimensional array of strings which stores records of the students enrolled in the course.

  - `roster` references the two-dimensional array, and
  - `size` keeps track of the number of student records stored in the array roster.

- A student record is represented by a one-dimensional array of strings (**String[ ]**) whose length is 3. The entries in every student record (`roster[i]`), are defined as follows, where **i** is the order of the student record in the roster:
  - **roster**[**i**][**0**] stores the ***name*** of the specific student.
  - **roster**[**i**][**1**] stores the ***email address*** of the student.
  - **roster**[**i**][**2**] stores the ***10-digits campus ID*** of the student.
- We assume that all the entries are valid and correctly formatted.

- We also assume that ***campus IDs*** are **unique**. We are going to use campus ID as a key search to find a matching student record.

- This is an oversize array, meaning that ALL the student records stored in the roster array **MUST** be in the range ***0 .. size-1***. These MUST be ALL **NOT null**. Any **null** reference **MUST** be in the range ***size .. roster.length-1***.
- Adding at the end of an oversize array means adding at index **size**, the first non-null position, if the array did not reach its capacity.
- Adding and removing operations **MUST** keep the order of precedence of the elements already stored in the array.
- This means that adding and removing from the beginning or the middle of the roster array must involve a shift operation.
- Make sure to review the first chapter on zybooks to get a better understanding of how oversize arrays are used within methods in java.
- **Sample of the contents of a roster:** Let's consider three students with the following details:
  - **name:** Mouna, **email:** mouna@wisc.edu, **campus id:** 1234567890
  - **name:** Michelle, **email:** michelle@wisc.edu, **campus id:** 2345678901
  - **name:** Mark, **email:** mark@wisc.edu, **campus id:** 5678901234

  A roster whose capacity is 5 and stores the above student records is defined as follows.

```
String[][] roster = new String[][]{
  new String[]{"Mouna", "mouna@wisc.edu", "1234567890"},
  new String[]{"Michelle", "michelle@wisc.edu", "2345678901"},
  new String[]{"Mark", "mark@wisc.edu", "5678901234"},
  null, null
};
int size = 3;
```

2.2 Waitlist to store the records of waitlisted students who did not secure a spot in the course

- **String[ ][ ] waitlist:** a ***non-compact* perfect-size** two dimensional array which stores records of students who are waitlisted. This array is defined by its reference ONLY. It does NOT define an `int` variable to keep track of its size.
- This is a ***non-compact*** array, meaning that null references at any index within the array.
- This 2D array stores one dimensional arrays of strings with student records in the same format as described above.
- A student record name, email, campusId can be added to the first null reference in the waitlist array.
- To remove an element off the waitlist, set its reference to null.
- The waitlist array is full if it contains NO null references.
- While traversing the array waitlist be sure to skip any null reference.

READ carefully through the details provided in the javadoc method style headers of the `CourseEnrollment` and `CourseEnrollmentTester` classes. Do not hesitate to ask for clarification if you notice any ambiguous specification. Then, proceed to the following implementations.

3 Implement and Test the Administration Tools Functions

We recommend start implementing the methods related to the administration tool functions and their tester methods. This includes the list of the following methods. **CourseEnrollment class methods:**

```
public static String[][] createEmptyList(int capacity) {}
public static void printRoster(String[][] roster, int size) {}
public static void printWaitlist(String[][] waitlist) {}
public static int indexOf(String campusId, String[][] list, int size) {}
```

```
public static int indexOf(String campusId, String[][] list) {}
```

CourseEnrollmentTester tester methods:

```
public static boolean createEmptyListTester(){/* provided */}
public static boolean indexOfPerfectSizeArrayTester() {}
public static boolean indexOfOversizeSizeArrayTester() {}
```

Test your implementation and fix any detected bugs before you proceed! Note that you do NOT need to test any displayed output to the console in your tester methods.

Note that the **sample run** provided at the last section of this assignment provides examples of displayed output of `printRoster()` and `printWaitlist()` methods.

4 Implement and Test the Enrollment and Waitlist Option Functions

Now, implement the following methods and their testers. Keep in mind to follow the advice on how to approach an algorithm provided at the top of this write-up.

CourseEnrollment class methods:

```
public static boolean addWaitlist(String name, String email, String campusId,
     boolean prerequisiteSatisfied, String[][] waitlist) {}
public static int enrollOneStudent(String name, String email, String campusId,
     boolean prerequisiteSatisfied, String[][] roster, int size, String[][] waitlist) {}
public static int dropCourse(String campusId, String[][] roster, int size) {}
```

CourseEnrollmentTester tester methods:

```
public static boolean enrollOneStudentTester() {/* provided */}
public static boolean enrollOneStudentMoveFromWaitlistTester{}
public static boolean successfulDropCourseTester() {}
public static boolean unsuccessfulDropCourseTester() {}
```

- Recall that ALL string comparisons in this program are CASE SENSITIVE.
- Feel free to re-use the provided helper methods to compare the contents of actual and expected perfect and oversize arrays.
- You can use **Arrays.deepEquals()** method to compare the expected and actual arrays defined in your tester methods.
- You can use **Arrays.deepToString()** method to get a string representation of the contents a two-dimensional array.

- Feel free to define private static helper methods in your Tester class.
- Test your implementation and fix any detected bugs before you proceed! Note that you are encouraged but NOT required to implement additional tester methods. This is the first programming assignment and we would like to introduce you to how to write tester methods.

For the methods that you are NOT implementing tester methods, we encourage you to write down self-check questions you think you might need to answer to make sure the operation to implement works well. Then, try to answer those questions. For instance,

- What would happen if an attempt is made to enroll a student already enrolled?
- What would happen if an attempt is made to enroll a student already waitlisted?
- What would happen if an attempt is made to enroll a student not satisfying the pre-requisites of the course?
- ...

You can also refer to the **sample run** available at the last section of this assignment to have a better understanding of the expected behavior of this program.

5 Driver Class

The final step in this assignment is implement `CourseEnrollmentDriver` class and run your program! A starter code for this class was provided at the *Getting Started* section of this write-up.

Read through the provided code and complete the missing statements marked by the `TODO` tags. This driver class allows the user to launch and run the Course Enrollment System application. Running the main method in this class must result in an interactive session comparable to the one illustrated in the sample run provided in the following section.

- You do not need to worry about erroneous input from the user. All of our grading tests will focus on properly encoded commands, as described within this specification.
- **NullPointerExceptions** and **IndexOutOfBoundsExceptions**, if thrown, indicate bugs in your program. You might need to review your code and fix them. Be sure to not use an array whose reference is null, or access an element at an out of bounds index.
- You MUST create and use only one instance of the Scanner class in your entire program.
- The following methods are useful while processing the user command lines:

  - String.charAt(index) – returns the character at the specified zero-based index from within this string.
  - Boolean.parseBoolean(String s) - returns the boolean represented by the string argument.
  - String.split(): You can split the command line arguments using String.split(" ") and get the array of strings storing each argument of the command line in order. You can also use string.split(":") to split a student record information entry around the colon separator and get an array of strings storing each piece of information in order.

- White space is the command argument separator. Extra white spaces should be ignored while processing the user input command. We are not going to consider user command lines with extra white spaces in our automated testers.

- Colon : is used as separator to delimit the entries of a user-provided student record information in the following format.

```
name:email:campusId:prerequisiteSatisfied
```

- All commands are provided in the following command menu:

```
[1 <roster_capacity> <waitlist_capacity>] Create a new course enrollment
[2 <name>:<wisc_email>:<campus_ID>:boolean(true/false)] Enroll student
[3 <name>:<wisc_email>:<campus_ID>:boolean] Add student to waitlist
[4 <campus_ID>] Drop the course
[5] Print roster
[6] Print waitlist
[7] Logout and EXIT
```

- the boolean in the commands 2 and 3 indicates whether the course pre-requisites are satisfied or not.

6 Sample Run

A demo of a **sample run** of this program, once completed, is available **here**. Note that the user's input below is shown in green, and that the rest of the text below was printed out by the program.

7 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only THREE files that you must submit are:

- `CourseEnrollment.java`
- `CourseEnrollmentTester.java`
- `CourseEnrollmentDriver.java`.

Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of Due: **9:59PM CT on September 14**th. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline.