# CS5010, Fall 2019

## Assignment 4

Tony Mullen

a.mullen@northeastern.edu

•**Assignment 4 (Event Seating Management) is due by 6pm Monday, October 21**

## Submission instructions: We are no longer using Bottlenose for assignment submissions. All work should be submitted via your Github repo.

So far this semester, you have likely been working in the "master branch" of your Github repo. In industry, work is never carried out directly in the master branch. Instead, engineers will create a "topic branch" to work on their assigned tasks. When work is complete and thoroughly tested, the topic branch can be merged into master. For the rest of this semester, you will adopt this workflow, writing all your code in topic branches then merging into master when you're ready to submit your work.

**Before beginning work on this assignment**
Make sure all your local changes have been pushed to your remote repo before continuing. Then, confirm that you are starting from your master branch by running `git checkout master`. Ensure that you have the most uptodate files by running `git pull`. If this is the first time you're using branches in your repo, you won't see any changes. However, these two steps will become very important once you have multiple branches.

Create a new branch for this assignment by running the following command from your local repo:

```
git checkout -b assignment4
```

Check you are in the correct branch by running `git branch` or `git status`. Do this every time you start a work session or resume work after a break. You can switch branches by entering `git checkout branch_name` (no `-b` flag). Replace `branch_name` with the name of the branch you want to switch to.

Carry out all work for this assignment in your `assignment4` branch. This includes adding, editing, and committing **files**. To push to your remote branch, enter the following:

```
git push -u origin assignment4
```

**Submission instructions**

To submit your work, create a "pull request" to merge your topic branch into your master branch. Tag your codewalk TA as reviewer.

**Step 1:** Make sure all your work on your topic branch has been committed and all commits have been pushed to your remote repo.

**Step 2:** Open your repo in your browser. You will see a yellow notification box with the name of the branch you just pushed and a green "compare & pull request" button.

**Step 3:** Click the "compare & pull request" button. You will be taken to a new page called, "Open a pull request".

**Step 4:** Click the settings icon in the "Reviewers" tab to the right of the text box. In the popup window, search for your codewalk TA. Click on your reviewer's username to send them a request.

**Step 5:** Click "Create pull request" to complete the process. You should see a confirmation that a review has been requested.

After grading, your TA will approve your changes. At this point, you will be able to close the pull request and complete the merge by clicking on the "Merge pull request" button. Do not merge the branch until you have received a review from your TA.

You can read more about pull requests in the Github docs:
https://help.github.com/en/articles/merging-a-pull-request

## Submission Requirements

The technical criteria for submission are the same as for the previous assignment. Namely:

- **Naming convention:** Your package name should follow this naming convention assignmentN, where you replace N with the assignment number, e.g., all your code for this assignment must be in a package named assignment4.
- **Gradlebuild:** Your project should successfully build using the provided build.gradlefile, and it should generate all the default reports.
- **Javadoc generation:** Your Javadoc generation should complete with no errors or warnings.
- **Checkstyle report**: Your Checkstyle report must have no violations.
- **Code coverage report:** Your JaCoCo report must indicate 70% or more code coverage per package for "Branches"and"Instructions".
- **Methods `hashCode(), equals(), toString()`:** all of your classes have to provide appropriate implementations for these methods. (It is sufficient to autogenerate these methods, as long as autogenerated methods suffice for your specific implementation). Please don't forget to autogenerate your methods in an appropriate order -starting from the ancestor classes, towards the concrete classes.

- **Javadoc:** please include a short description of your class/method, as well as tags from @paramsand @returnsin your Javadoc documentations (code comments). Additionally, if your method throws an exception, please also include a tag @throwsto indicate that.
- **UML diagrams:** Please includeUML diagrams for the final versions of your designs for every problem. In doing so, please note that you do not have to hand-draw your UML diagrams anymore. Auto-generating them from your code will be sufficient.

## Problem: Event seating management

You have been tasked with writing a management system for automatically generating seat assignments for seating at conference/convention events attended by people in multiple large groups. Each attendee to an event should be given a section, row, and seat (column) assignment.

You are **not** expected to generate a globally optimal seating arrangement, but your system should do a reasonable job of obtaining a desirable seating arrangement according to the specifications below.

Assume you are working as a back-end developer, and are free to define the API as you see fit. However, some design decisions (such as the "group" field described below) have been made by the front-end team and are not within your control, so you will need work with what is available.

### Venue layout

The system must work for a variety of venues. All venues' seating is assumed to be broken up into rectangular sections which are in turn broken up into numbered rows and seats. A venue may have any number of sections. In most venues, all the sections are the same dimensions, but it is possible that a venue might have variously sized sections (that is, sections with differing numbers of rows and seat-columns). All sections in all venues may be assumed to be rectangular (that is, having a consistent number of seats per row throughout the section). You can assume that sections in a venue are always laid out in a grid.

### Groups

It is assumed that attendees will attend the events in groups representing, e.g. schools, towns of origin, companies, etc. It is important that attendees are seated as near as possible to other attendees from the same group. Attendees should not be seated in an isolated seat away from the rest of their group, although large groups may be divided and seated in separate smaller groups. Seating in adjacent sections (e.g. across an aisle) may be regarded as nearby to each other. In general, the definition and implementation of "proximity" is up to you.

Group self-identification is submitted as part of a reservation request (see below for details). Each group has a group leader, who may book multiple tickets for the group and who receives updates when group members' seating assignments change.

### Wheelchair accessibility

Attendees may require wheelchair accessible seating. All venues must have wheelchair accessible seats in each section. Most commonly, the first row of a section is made up of wheelchair accessible seats (you may assume that accessible seats occupy the same dimensions as other seats) but it is possible in principle that any seat in the section may be designated as a wheelchair accessible seat.

Wheelchair accessibility is always a priority. Wheelchair accessible seats should only be assigned to those who don't require them after all other seating options have been exhausted. Wheelchair users should never be assigned non-accessible seats.

In the case where an attendee would otherwise need to be seated alone away from their group, it is acceptable to assign that attendee to a wheelchair accessible seat, even if they do not require one.

## Input

Reservation requests may be submitted either for individual attendees by themselves, or for collections of attendees in a single group, by the group's leader. You do not need to worry about front-end authentication for reserving; you may assume that any requests have arrived through some authenticated channel and that the request is allowed. In the case of individual attendee requests, the attendee's group affiliation is passed as part of the request along with their name. In the case of group reservations, a list of attendees is submitted, all of whom are members of the same group, along with the group name they belong to. Reservations are not exhaustive; more than one individual or group reservation can come in for attendees of the same group, and register requests may continue to come in up to the last minute. Newly registered attendees should be seated near members of their group.

## Group field input

Reservation requests by either attendees or representatives of attendees are submitted with a corresponding group associated, as described above. However, the group field for the submission form (which you as a back-end developer have no control over) is implemented as a free text field (limited to 40 characters). This means that there are sometimes inconsistencies in attendees' self-identified groups. For example, attendees associated with the same group may label their group as "Murray High School", "Murray Senior High", "Murray HS" or even "the murray group". It is probably not possible to attain perfect accuracy, and the web-development team is working on a more restrictive interface, but for now this is what you have to work with. You should at least try to scrub input of extraneous words like "group", "school" or "company", and normalize capitalization. Regular expressions may be helpful for this.

## Seating changes

In order to best optimize the seating arrangements with respect to wheelchair accessibility and group member proximity even as an unpredictable number of people registers, the seating plan for all attendees remains dynamic until individual attendees "print" their ticket ("print" could simply mean downloading a finalized e-ticket. Think of this step as analogous to getting a boarding pass on a plane). Once the "print ticket" method has been called for a given seat, that seat must be regarded as fixed and can no longer be re-assigned for the above purposes.

## Group leader notifications

Group leaders are updated on the current seating plan for members of their group, even if it is not finalized. Group leaders receive updates any time changes occur that affect the seating plan for members of their group.

Group leaders must also be notified if any requests for reservations for their group members fail.

You do not need to implement the actual notification, but you should ensure that a method is called that will trigger the notification process for the appropriate group leader.

Use the observer pattern to implement group leader notifications.

## Summary of required functionality

The system should be able to handle the following interactions to the extent described above:

- Reservation request for individuals or lists of attendees of a common group. The system should maintain a seating arrangement that is updated on each reservation request. Reservations should fail with an informative response in the following situations:
    - The venue is full
    - Attendees who require wheelchair accessibility are not able to be assigned a wheelchair accessible seat
    - The only seating available to an individual is isolated from (i.e. not adjacent to any member of) that individual's group, despite reasonable attempts at re-arrangement (you determine what is reasonable).
- Cancellation request for individuals. The system should update its seating arrangement for each cancellation.
- Ticket "printing" request for individuals. You do not have to implement ticket printing, but you must write a method that will represent triggering a print ticket process, after which that specific seat must not be reassigned or canceled.
- Group leader notification.

## Your tasks

1. Create whatever Java classes, subclasses, abstract classes, and interfaces appropriate for representing venues, groups, attendees, and seating arrangements. Be prepared to justify your design decisions.
2. Implement the reservation request and seat assignment logic that takes into consideration wheelchair accessibility, seat availability, and group proximity. You will need to determine a useful proximity metric yourself. If you encounter clashes of priorities not explicitly covered in the specifications, you will need to handle these in a sensible way.
3. Write tests to cover new functionality. Include tests with a variety of seating states to ensure that newly seated attendees are seated with their groups, seating is updated when necessary as new attendees join, and that group leaders are notified if seating arrangements change.
4. Provide your final UML diagram that includes all relevant methods.