# CSE 331, Spring 2011
# Homework Assignment #3: Restaurant (50 points)
# Due Friday, April 22, 2011, 8:00 AM

This program focuses on class design, pre/postconditions, invariants, specifications, and Javadoc commenting. Turn in the Java files listed below from the course's homework web page. It is an **individual** assignment, though you can have one "buddy" with whom you may discuss the details of your design (see the "Design Buddy" section of this document).

You will need to download support files like `RestaurantMain.java` , `RestaurantTextUI.java`, and `tables.txt` from the course web site into your project to run the GUI. Run `RestaurantMain` to launch the program. Note that input files like `menu.txt` should be placed in your Eclipse project's root folder, the *parent* of its `src/` or `bin/` subdirectories.

## Program Description:

In this program you will write a set of supporting classes for a restaurant management system. This kind of application would be used by the hosts and servers at a restaurant to manage their set of tables, customers, meal orders, employees, and so on. The instructor has written a text user interface that will provide the "front end" or "view" to your program. (Past assignments supplied a graphical user interface. But in this assignment, you will have to make some modifications to the UI, so we are providing a text interface so that its code will be more familiar.)

*There are many real-world aspects of running a restaurant that are not represented in this system.* For example, the system does not know the customers' or servers' names, nor does it know the menu of food choices available at the restaurant. The system focuses on allocating parties to tables, handling the waiting list, and processing payments and tips.

## Classes to Implement:

- `Restaurant`, the overall restaurant and its associated data
- `Party`, a group of customers that can eat at the restaurant
- `Table`, a table at which parties of customers may sit to eat
- `Servers`, a manager for servers and their allocation to tables in the restaurant
- `Utility`, a place to store shared helper code (optional)

This document does not specify exactly what behavior, fields, methods, etc. each class should have. Part of this assignment is for you to choose appropriate contents for each class based on the specification of desired functionality and based on our class discussions of proper class design. A major part of your grade will come from how effectively you design the contents of each class to solve this problem elegantly.

### *Restaurant:*

The restaurant as a whole is represented as an object of your `Restaurant` class. This is a central class that manages several of the other pieces of data and objects in the system as well as helping serve as a bridge of communication between other objects in the system. The restaurant has many attributes, such as its tables, its waiting list, and so on.

Be careful, though, not to place too much functionality into the `Restaurant` class. Follow the Expert pattern to place functionality into other objects as appropriate when those other objects possess more of the data or knowledge needed to perform the action. If your `Restaurant` class is very large and the others are very small, you have designed poorly.

This program is not especially concerned with the passage of **time** or the current time of day. For example, you don't need to worry about the time that the restaurant opens and closes, the time that a party of customers arrives to eat, the time needed to cook their food, etc. As the application is running, your code should operate under the assumption that the restaurant is currently open and will remain open while the app is executing. We will also simplify our program by assuming that the restaurant does not worry about parties who want a spot in line to eat at a later time; in other words, the restaurant does not take **reservations**.

### *Tables:*

The restaurant has a numbered collection of **tables**. Each table is represented by an object of your `Table` class. Each table can fit a specific number of customers. The number and sizes of tables are read from an input file; a sample input file is provided. You should provide a "producer" method that reads a file of table data in the provided format and uses that data to populate the collection of tables at a restaurant. The file's expected format (see Exceptions section) will consist of an initial line containing the restaurant's name, followed by a second line containing a list of whitespace-separated integers representing the sizes of the tables in the restaurant. For example:

```
Joe's Restaurant
4 4 8 4 2 4 2 6 4 2
```

As customers enter and leave the restaurant, tables become either occupied or unoccupied.

### *Parties of Customers:*

Tables at the restaurant can be occupied by **parties of customers**; each party is represented by an object of your `Party` class. (We could choose to model each individual customer, but for the purposes of this application, the details of each customer such as their name, age, order, etc. are unimportant.) The restaurant thinks of parties in terms of a party name and how many people are in the party; for example, the Smith party of 5 or the Jones party of 2. Party names are unique; no two parties in the restaurant at the same time can have the same name. The program should forbid a party that attempts to be seated or placed onto the waiting list with a name that exactly matches an existing table's name.

If a new party of customers arrives and asks to be seated, and a table large enough to accommodate them is available, the party will be seated immediately. The party will be seated at the **smallest available table** that can accommodate the entire party. For example, suppose a party of 5 arrives and the various tables seat 2, 4, 6, or 8 customers. A 6-customer table would be preferred, or if none was available, the party would be seated at the 8-person table. If there is a tie in terms of table seats, the lower-numbered table of that size will be preferred.

To simplify our code, if a party arrives with more people than the largest table at the restaurant, they are **refused service**. For example, if the largest table in the restaurant seats 12 people and a party of 15 arrives, they are not served. (Sad!)

Sometimes a party at a table doesn't occupy all of that table's seats, but we will simplify our program by assuming that at most one party can sit at a given table. As an extreme example, a table with 12 seats that has a party of one (a single person) sitting at it will be considered to be **fully occupied** for the purposes of this program.

### *Waiting List:*

If no table large enough to accommodate the arriving party has arrived, the party is put on a first-come, first-served **waiting list**. When an existing party leaves the restaurant, their table becomes free. If there are parties on the waiting list, the party closest to the front of the line that will fit at the newly freed table is seated there. For example, suppose the waiting list includes parties of 5, 2, 6, and 3 people. If a table opens up that has a capacity of 4, the first party in line (with 5 people) can't fit, but the party of 2 will, so they are seated and the waiting list now consists of the parties of 5, 6, and 3. Next, if a table opens up that has a capacity of 8, the party of 5 will be seated there. And so on.

Note that it is possible for there to be a nonempty waiting list but still have an arriving party be seated immediately. For example, if parties of 5, 4, and 7 are waiting because no table that large is available, but a party of 2 arrives and a table of size 2 is available, the party of 2 is seated immediately.

### *Servers:*

The restaurant has a group of **servers**, a.k.a. waiters, who are the employees who serve the parties of customers. This group is represented and managed by an object of your `Servers` class. The `Servers` object should keep track of all server-related details and should contain most or all of the major behavior related to servers and their interaction with the other parts of the restaurant as appropriate. Initially the restaurant has no servers, but as servers arrive to work for the evening, the system can be notified to add them.

We will simplify our program by not worrying about excessive personal details about each server, such as the server's name, age, etc. Servers are identified solely by ordinal **numbers**; the first to arrive for the evening is Server #1, the

second is Server #2, and so on. Servers can arrive at any time, and each newly arriving server is given a number value 1 higher than the highest-numbered server that has ever checked into the system during this session. If no servers have yet arrived at the restaurant and been entered into the system, no tables of customers may sit to eat.

When a newly arriving party is seated at a table, a server is assigned to them. The servers are allocated to tables in a **round-robin** fashion. Suppose 3 servers are working. The first party to arrive and be seated is assigned to Server #1, the second to Server #2, the third to Server #3, the fourth back to Server #1, the fifth to #2, and so on. The main importance of assigning servers is to make sure that the **tip money** left by the party will be paid to that server. Note that the server is assigned to the party as they are seated; if the party is put onto the waiting list, a server is not yet assigned to them until they exit the waiting list and are seated at a table.

### *Paying the Check:*

When a party is finished eating, the system records their **payment for their food**. The system asks for the subtotal cost, which is the total cost of the food ordered by that party without considering any tax or tip. The system then applies a **10% sales tax** to the subtotal. The system also asks for the party's tip. Parties leave a **tip** of some amount of extra money as they leave the restaurant to reward the server; the tip is given directly to the server. The rest of the payment (subtotal plus tax) is given to the restaurant itself, placed into the restaurant's **cash register**. Tips are not taxed and not put into the restaurant's cash register. At any time the restaurant management software can display a total of how much money each server has earned in tips so far, as well as displaying the total amount of money in the restaurant's cash register.

Once a party pays their check and leaves a table, the **table becomes unoccupied** and available for further use. The server is also released from his/her allocation to that table. A new arriving party could potentially now be seated at the table. If there is any waiting list when a table becomes available, the party closest to the front of the waiting list that can fit at the newly open table is seated there immediately. When a new party is seated at the newly available table, the server who is currently next of the round-robin ordering will be allocated to serve that table.

### *Servers "Cashing Out" at the End of the Night:*

At any moment a server may be released from work for the day to **go home**. The only server who may be dismissed is the one at the front of the round-robin ordering (the one who would have been the next chosen to serve a new party that were to come in). If a server is released while that server is still allocated to parties at tables, those tables are reassigned to the other servers in round-robin order. For example, if the round-robin ordering of servers goes 2-3-1, and server 2 is sent home, server 3 gets server 2's first (lowest-numbered) table, server 1 gets the second table, server 3 gets the third, 1 the fourth, and so on. These newly reassigned servers will be the ones who receive any tips that come from the tables to which they have been reassigned. If only one server remains and there are still parties dining, that server may not cash out until all parties have paid their checks and left.

When a server is released, he/she will "**cash out**" her prior tips and be removed from the round-robin ordering for the rest of the evening. The servers are paid entirely by their tips and receive no other salary.

### *Text User Interface:*

The classes in the preceding list should not perform any **user interaction**. That is, there should not be any `System.out.println` statements or `Scanner` input from the console performed directly by these classes. Instead, you should have the provided `RestaurantTextUI` class perform all user input/output.

The text UI given to you is mostly complete, but it is missing a few "hooks" into your code. We aren't able to write in exactly what calls the text UI should make because we don't know in advance exactly how you will design your classes. So **you must edit the provided mostly-complete text UI class**, looking for places that are unfinished and adding small bits of code there to attach the text UI properly to your objects and their methods. These parts of the code have been marked with `// TODO` comments to make them easier to find.

**Do not place core system functionality into the text UI class**. Its primary purpose should be to read input from the user and then hand off this input data to your other core objects as described previously, to let those core objects do the bulk of the actual work and computations. For example, it is not appropriate for the text UI class to contain the code that determines which server should be assigned to which table. You may add fields and methods to the text UI class as

necessary or desired, but the text UI should not directly store any of the core data of the system. For example, it is not appropriate for the text UI to store an array of all the tables in the restaurant.

Your classes are to exactly reproduce the format and overall behavior found in our sample solution and sample outputs. In several places the text UI displays amounts of **money**. A given amount of money should always be displayed by the UI preceded by a dollar sign and with exactly 2 digits after the decimal point, such as $0.60 or $12.00 or $34.56.

You will have to run the text UI and test individual inputs on your own to verify that your classes are working correctly. A **sample solution** will be posted to the course web site that you can run to verify the behavior your program should have. The sample solution is our best effort to try to meet this spec, but it might contain minor bugs or errors. If the sample solution's behavior differs from this specification, follow this spec, not the sample solution. Please ask if you are unsure.

## Exception Checking and Error Handling:

Your classes should forbid invalid parameters. If any method or constructor is passed a parameter whose value is invalid as per this spec, you should throw an `IllegalArgumentException`. In particular, you must enforce the following:

- No parameter passed to any method (except `equals`) should ever be **`null`**.
- Strings that represent names should not be **empty** or consist entirely of **whitespace**.
- Integer fields that store counts of real-world things or persons, such as the number of people sitting at a table, should always be **non-negative**.
- Prices and amounts of money (e.g. subtotals, tips) should always be **non-negative**.
- If you parse data from files as strings and turn the strings into objects, you should check that the strings are in a **valid format** and throw the exception if they are not.

The main client program should not throw any **exceptions** to the console. If any errors occur, it is considered a bug in your code. It is okay (encouraged) for you to incorporate exception throwing into the design of your core classes, but you should make sure to catch those exceptions in the text UI client or somewhere else in the active call stack.

The text UI performs its own checking for the validity of various user input values, such as making sure that numbers are non-negative. But this is completely unrelated to your own tests and exception checks. Remember that valid arguments are preconditions for your methods, and preconditions are *supposed* to be things that the client could check for and avoid. Just because this particular client does so does not absolve you from performing checks in your core classes' code.

## Design "Buddy":

This is considered an individual assignment; you must submit your own solution that you implemented by yourself. But to help you compare ideas for good design, you are allowed to have one optional **"design buddy"** for this assignment. Your "design buddy" is one other student in the class with whom you are allowed to speak in great detail about the designs you both have chosen for your classes. In other words, you may share with your buddy the exact methods, fields, etc. that you have chosen to use to implement this specification along with the reasoning behind those decisions, pros/cons you have discovered about your implementation of those decisions, etc. The goal behind this is to allow you a limited amount of greater collaboration, to help make the assignment more manageable, and for you to learn from each other.

If you choose to have a "design buddy," each of you should document this in your code by writing your buddy's name in your class header comments. (It is generally good practice to cite sources and collaborators in submitted code.) You should also include the design buddy in the `@author` tag of any Javadoc comment headings atop your classes.

## Development Strategy and Testing:

If you try to write all of these classes at once and then compile/run the overall program, you are unlikely to succeed. This is a large system that uses all of your classes in complex ways. A small error in your code will stop it from functioning entirely, giving you poor feedback about what code does and does not work successfully.

Therefore it is important to write and test your code incrementally. We suggest that you initially write the smaller, simpler classes that have less behavior and fewer connections to other classes. Remember that you can insert a "stub"

version of a particular method or entire class that simply has a pair of empty `{}` braces for the method's body (or simply returns a dummy value like `null` or `-1`). This may allow you to test unfinished classes together.

You are not required to turn in a test file as you did on Homework 1, but we strongly recommend that you write small testing program(s) anyway. You can verify most of the basic functionality of these classes with just a few lines of testing code. This is simpler than trying to jump right into the GUI, which requires everything to be done before it will run.

**Please do NOT share any testing program code you write for this assignment**, such as JUnit tests or `main`-method test programs. The reason is that your testing code will reveal details about the implementation and design you have used, through the set of methods it tries to call on your objects, and we do not want your design to be revealed to other students. You may, however, share text input files or input/output logs that you have created from your runs of the main text UI, to help show illustrative usage of the client that helps to find flaws and bugs.

## Style and Design Guidelines:

Since you will be designing the contents of the classes for this assignment, a large part of your grade will come from the quality of your **class design**. Follow guidelines taught in class such as: striving for the "C" words (cohesion, completeness, clarity, convenience, and consistency); minimizing unnecessary coupling; choosing a good set of methods for each class; choosing a good set of fields for each class; designing constructors to accept appropriate parameters to properly and fully initialize the object and minimize their workload; following standard naming conventions; making all classes `final` that are not to be extended by inheritance; following the Expert pattern, the Law of Demeter, and the Open-Closed Principle; and avoiding representation exposure.

Your code will be graded on whether it follows the **style and design guidelines** taught in lecture, particularly those indicated as tips from the *Effective Java* textbook. You should also follow a readable and consistent style for your classes such as the one found in the Sun Java official coding conventions linked from the course web site.

Follow the **Expert pattern** throughout your code, making sure that the object that knows the most about how to perform a given task (or is most closely related to the idea of that task) performs that task. For example, if a given type is read and/or written from a data source such as a file, that type should be the expert in regards to how to read itself properly.

The majority of the methods you write should run in **constant time** (O(1)) regardless of any parameter value(s) passed. Methods that must process an entire data structure iteratively are by nature not O(1); this is expected and allowed for some parts of the functionality of the program, such as displaying all tables at the restaurant.

This document does not specify what **collections** to use inside of your various objects. But you should always choose data structures and algorithms that minimize the complexity class necessary for solving a given task. For example, you could choose to implement round-robin server scheduling by performing an $O(N^2)$ nested traversal of all servers, but this is unnecessarily inefficient. Choose the collection that is most appropriate and that will provide the fastest overall runtime for the operations implemented, taking into account any issues related to desired ordering, duplicates, mapping, and so on.

You should intelligently choose which behavior should and should not be declared **`static`**. The majority of your classes' behavior should be non-static, residing in the individual objects of each class; but methods that bear no relation to the data of any particular object of that class should be `static`. For example, heavyweight producer methods to create objects by reading external data sources are best implemented as `static` methods.

You should intelligently choose which behavior should and should not be declared **`public`**. If a given method is a natural part of how clients would expect to use that object, make it `public` in the interest of providing a clear, complete interface. If a method is used primarily by the class upon itself to help it implement other functionality, it should be made `private`.

As appropriate, your classes should provide standard Java methods such as `equals`, `compareTo`, and `clone`.

A major focus of this assignment is ensuring that all of your objects are always kept in a valid state. This is accomplished through rigorous **argument checking** and throwing exceptions on invalid arguments. Part of your grade will be based on whether you handle all of the exceptions and all possible combinations of invalid arguments that could be passed.

**Document** all of your files by commenting them descriptively *in your own words* at the top of each class, each method/constructor, and on complex sections of your code. You must use **Javadoc comments** for all external

documentation (all comments atop class headers or public method/constructor headers). Your Javadoc comments should include descriptive initial summaries along with proper tags.

Your **class header comments** should include this course (the assignment / course / section), author information (your name and other contact info), a description of the Java class itself (the overall purpose of the Java class written in the file), along with any class **invariants**. Think carefully about what invariants clients may assume about the objects you have written. Come up with a reasonable set that is useful to clients and consistent with the expected functionality of the restaurant; document those invariants, then enforce them rigorously throughout the class.

Your **method header comments** should describe the method's behavior, parameters, return values, exceptions thrown (the type of exception and what would cause it to occur), any modifications to the state of the current object ("this"), and any pre/postconditions. Private helper methods are graded more loosely on commenting than public ones; simply give a private method a brief header explaining its purpose.

You should also include **internal documentation** in the form of `//` comments written inside bodies of classes and methods as appropriate to describe complex sections of code. These comments are different than external comments; they are for a fellow developer or maintainer of your own classes, not for clients. Therefore these internal doc comments should describe what the code is doing and how/why it is doing it, as appropriate. Trivial code does not need such comments, but complex operations should receive at least a modest amount of internal documentation.

**Redundancy** is a major grading focus of every assignment for this course. If you find yourself repeating code, make the code into a method to reduce the redundancy.

Follow good **general style guidelines** such as: making fields `private` and avoiding unnecessary fields (you will lose points if you declare variables as fields that could instead be declared as local variables); declaring collection variables using interface types (e.g. `List` rather than `ArrayList`); initializing fields in constructors, not at declaration; appropriately using control structures like loops and `if`/`else`; following "Boolean Zen" (proper use of boolean logic); properly using indentation, good variable names and types; and not having any lines of code wider than 100 characters.

For reference, our solution contains roughly **210 "substantive lines"** (which excludes things like blank lines, comments, and `{}` brace lines) according to the class Indenter Tool, excluding our `Utility.java` shared code. But this number is just provided as a sanity check; you do *not* need to match it or be close to it to get full credit.