Assignment 2
COMP 250, Fall 2021

Posted: Wed. Oct. 13 2021
Due: Fri. Oct. 29 at 23:59 (midnight)

[document last modified: October 18, 2021]

General instructions

- Search for the keyword *updated* to find any places where the PDF has been updated.
- T.A. office hours will be posted on mycourses under the "Office Hours" tab. For zoom OH, if there is a problem with the zoom link or it is missing, please email the T.A. and the instructor.
- If you would like a TA to look at your solution code outside of office hours, you may post a *private* question on the discussion board.
- We will provide you with some examples to test your code. **See file Test.java which is part of the starter code.** If you pass all these tests, you will get at least 60/100. These tests correspond exactly to the exposed tests on Ed. ~~Unlike in Assignment 1, however, we will not use *hidden* tests on Ed. Instead, we will use *private* tests.~~
  [Updated Oct. 18: We will use hidden tests as we did on Assignment 1. So you *will* know how many tests you have passed. There are 40 points for hidden tests.]

  We strongly encourage you to come up with more creative test cases, and to share these test cases on the discussion board. There is a crucial distinction between sharing test cases and sharing solution code. We encourage the former, whereas the latter is a serious academic offense.
- **Policy on Academic Integrity:** See the Course Outline Sec. 7 for the general policies. You are also expected to read the posted checklist PDF that specifies what is allowed and what is not allowed on the assignment(s).
- **Late policy:** Late assignments will be accepted up to two days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Please see the submission instructions at the end of this document for more details.

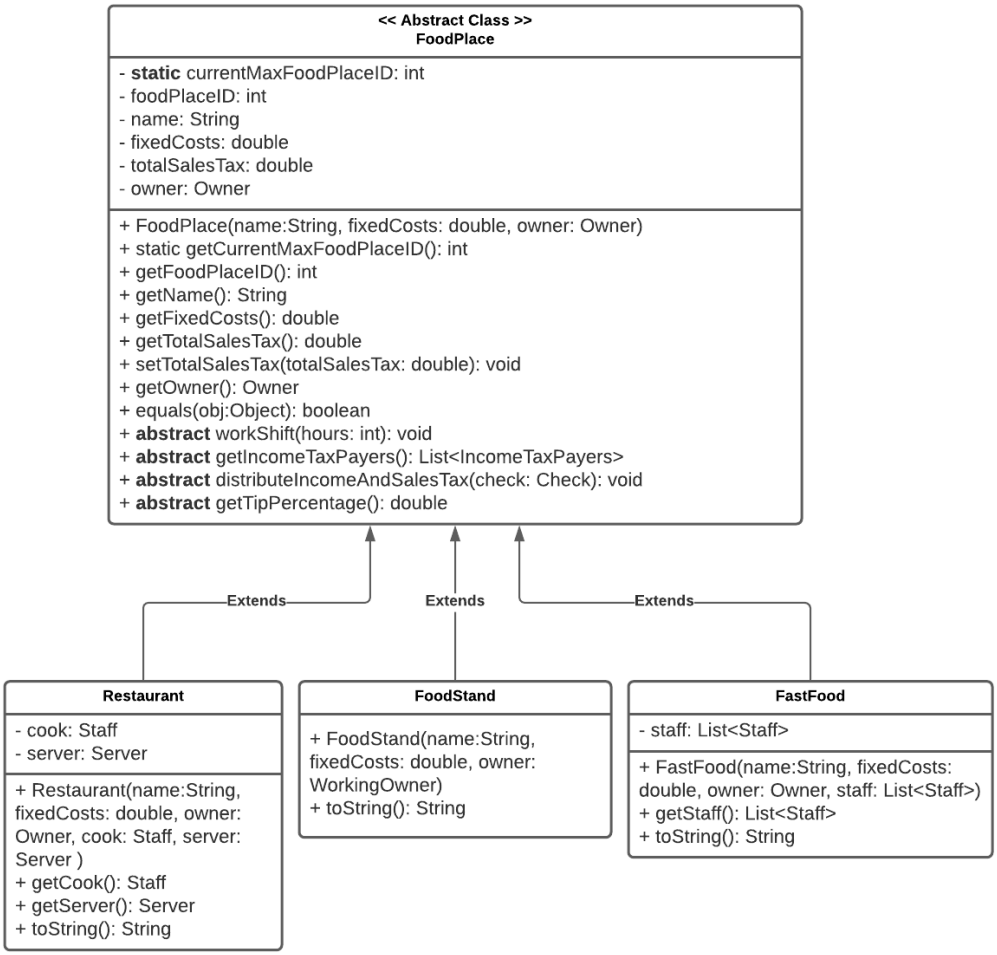
1 Introduction

In this assignment, you will get some basic experience working with inheritance and polymorphism. We model parts of a small economy, namely the food service industry. We model places where customers can buy food: restaurants, food stands, and fast food. We model the payment of bills, including sales tax and tip. We do not model many other things such as the food or menu. We also model a tax collector who collects both sales tax and income tax. We only model one year: the workers each have some income and pay tax for that year.

This document is organized as follows. In this section we present the various classes of the model. We chose the ordering of classes in the way that we think it easiest to understand. Later in the document (Sec.) we recommend a different ordering for you to implement and test the classes. Note that many of the methods have been implemented for you, in particular, all the getters and setters.

We present a UML diagram for each class, and indicate if one class extends another. For all the classes in this assignment, all fields are private and all methods are public. In the class descriptions below, we do not bother to mention fields or methods whose meaning is obvious, such as getter or setter methods.

We start with a few classes that model the food places. Here are a UML diagram for FoodPlace classes and subclasses.



FoodPlace [8 points]

FoodPlace is an abstract class. It has several private fields such as:

- foodPlaceID – each FoodPlace has a unique ID; the id's are determined by a counter, namely the following static variable (currentMaxFoodPlaceID)
- currentMaxFoodPlaceID – at any time, this is the largest ID of any FoodPlace; when a new FoodPlace is instantiated, the value of this variable is incremented and becomes the id of this new FoodPlace
- fixedCosts – every FoodPlace has certain maintenance, supply, and other fixed costs (lumped together)
- totalSalesTax - each Check paid by a Customer will have a sales tax that is added; the FoodPlace needs to accumulate these sales taxes; they will be collected by a TaxCollector.

You will implement the following public methods. *See the UML diagrams for the parameters of all methods.*

- a constructor FoodPlace() **[4 points]** - sets the fields; also, sets the owner's FoodPlace field.
- equals() **[4 points]** – returns true if the argument object is indeed a FoodPlace and the foodPlaceID's match; otherwise, it returns false

There are also abstract methods which will be described in the the subclasses below.

Restaurant [18 points]

The Restaurant class extends FoodPlace. You will implement the following public methods:

- a constructor Restaurant() **[4 points]** – sets the fields; note that a Restaurant is a FoodPlace, so call super().
- workShift() **[4 points]** – returns nothing (void). This method models that the cook and server are paid an hourly wage: specifically, the method increases the cook's and server's incomes based on the number of hours worked and per hour salary; the method also adds these wages to the owner's accumulated salary expenses.
- getIncomeTaxPayers() **[4 points]** – returns a List of IncomeTaxPayers that work at that Restaurant, namely a cook, server, and owner; the List return type allows you to use either a LinkedList or ArrayList (either is fine). The order of IncomeTaxPayer's in the list doesn't matter.
- distributeIncomeAndSalesTax() **[4 points]** – returns nothing (void) ; the parameter for this method is a Check. In all restaurants, the menu price component of the check is added to the owner's income; the tip component is added to the income of the cook (20 % of tip) and to the server (80 % of tip); the total sales tax is accumulated as well.
- getTipPercentage() **[2 points]** – returns the Server's target tip percentage

FoodStand [10 points]

The FoodStand class extends FoodPlace. You will implement the following methods:

- a constructor FoodStand() **[2 points]** – sets the fields; note that a FoodStand is a FoodPlace, so call super().
- getIncomeTaxPayers() **[2 points]** – returns a List containing one element (the owner); it can be either a LinkedList or ArrayList – your choice!
- distributeIncomeAndSalesTax() **[4 points]** – using the check, it updates the owner's income by adding the menu price and the tip; it also updates the total sales tax.
- getTipPercentage() **[2 points]** – returns the target tip percentage defined by the owner; note that (1) the owner of a FoodStand is a WorkingOwner which has a field targetTipPct, and (2) this method requires casting.

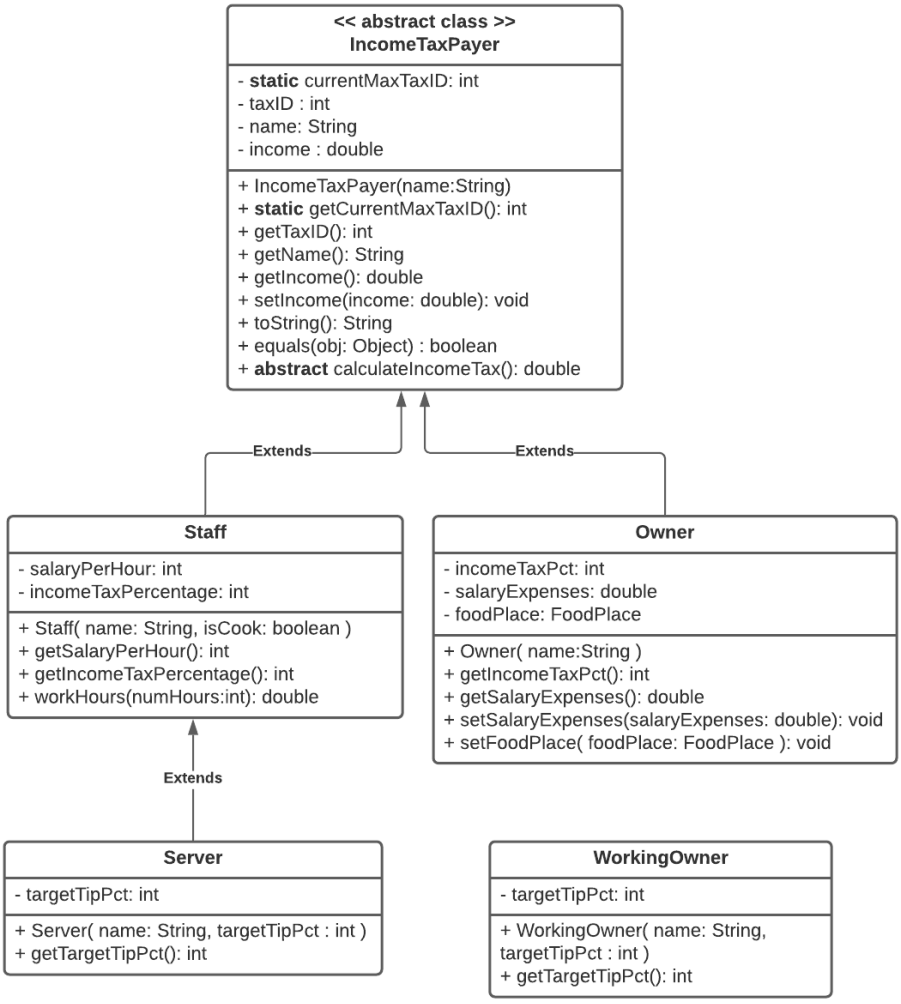
FastFood [14 points]

The FastFood class extends FoodPlace. You will implement the following methods:

- a constructor FastFood – **[4 points]** sets the fields; note that a FastFood is a FoodPlace, so call super(). The Staff list field must contain a shallow copy of the Staff list argument.
- workShift() **[4 points]** – returns nothing (void) ; it models that the owner pays each staff an hourly wage. The method adds an amount to each of the staff's incomes, and it adds these amounts to the owner's total salary expenses. Hint: use the Staff.workHours() method to add to a staff's income.
- getIncomeTaxPayers() **[2 points]** – returns a list (ArrayList or LinkedList) that includes the owner and all the staff in any order. Return a shallow copy of the list here.
- distributeIncomeAndSalesTax() **[4 points]** – returns nothing (void); in all fastfood places, the menu price is an income for the owner. It is also possible to have tips, but the tip only depends on the customer. The tip is split equally between all the staff (which does not include the owner). The totalSalesTax must also be set.

You do not need to implement the getTipPercentage() method since it is trivial: it returns 0. However, note that this only corresponds to one factor that affects the real tip. The customer is the second factor. So for FastFood, only the customer determines the tip set on a check. See the Customer.dineAndPayCheck() method for details.

Let's next look at the various types of people who work in these food places. These people all are income tax payers.



IncomeTaxPayer [6 points]

This is an abstract class. You will implement the following methods:

- constructor IncomeTaxPayer() **[4 points]** – assigns a uniqueID to this income tax payer; just increment a static counter currentMaxTaxID here to get unique id's.
- equals() **[2 point]** – returns true if the argument is indeed an IncomeTaxPayer and if the id's match, and otherwise returns false. Note that two IncomeTaxPayers might have the same name, but they cannot have the same id's. So you don't need to check the names.

Staff [8 points]

Staff is a class which extends IncomeTaxPayer. You will implement the following methods:

- a constructor Staff **[2 points]** – the salaryPerHour is determined by the parameter isCook: a cook is paid $20 per hour and otherwise a staff is paid $10 per hour.
- workHours() **[4 points]** – returns the amount of money earned by that staff, and modifies that staff's income.

  The parameter of this method is the number of hours worked in a shift; whenever a staff works this number of hours, the staff's income increases; this method will be called in the different implementations of FoodPlace.workShift() method. Note that this method is both a mutator and an accessor, namely it changes a field in the staff object and it also returns the amount of money earned by that staff.
- calculateIncomeTax() **[2 points]** – all staff pay 25% income tax.

Server[2 points]

Server is a class which extends Staff. (Note that Restaurant is the only FoodPlace that has a Server.) You will implement one method:

- a constructor Server() **[2 points]** – note that a Server is a Staff and in particular a Server is not a cook.

Each server (and cook) receives a tip from each Customer as part of the check – see the Restaurant's distributeIncomeAndSalesTax() method. The tip is calculated by the Customer.dineAndPayCheck() method: the tip depends on both the Server (e.g. their skill and friendliness) and on the Customer (whether they are generous or not), namely the final tip percentage is the average of the target tip percentages of the Server and the Customer.

Owner [6 points]

The Owner class extends IncomeTaxPayer. You will implement the following methods:

- a constructor Owner() **[2 points]** – sets the name of this owner (but note that owner inherits[1] its name field from its superclass)
- calculateIncomeTax() **[4 points]** – returns the amount of tax that the owner needs to pay, based on the owner's income and expenses. The owner's income is the sum of the menu prices on all the customer's checks. The owner's expenses are the hourly wages paid to the staff (salaryExpenses), and certain fixed costs (fixedCosts) for running the restaurant (e.g. maintenance, equipment, food supplies, etc) which we do not model in detail. The owner's income tax is 10 percent of the difference of the income and the expenses, assuming the income is greater than the expenses; if the income is less than the expenses, then the income tax is zero rather than negative.

[1] see lecture 13, slide 6

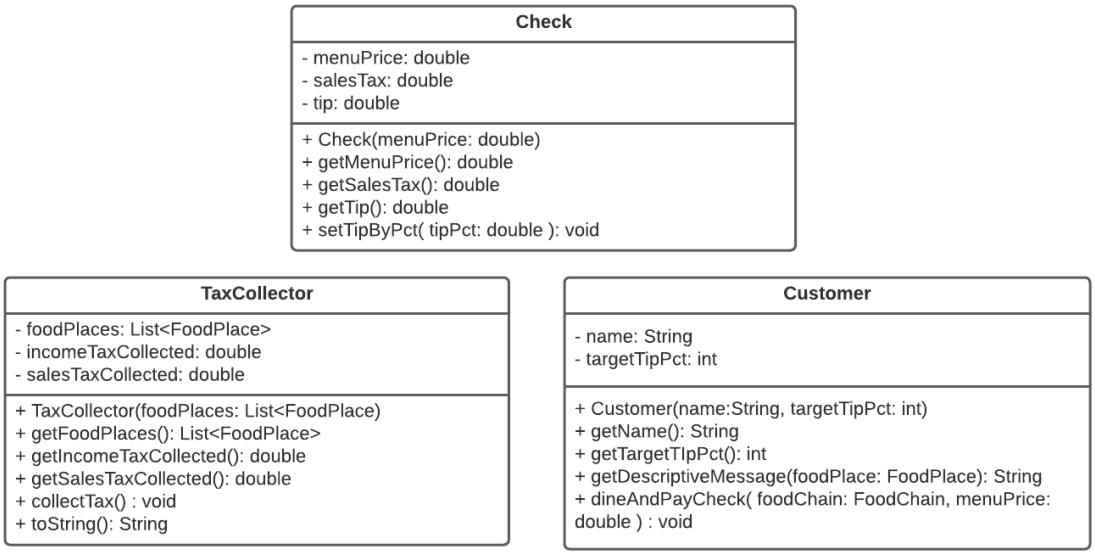
WorkingOwner [4 points]

WorkingOwner extends Owner. In our model, the only WorkingOwners are the owners of FoodStands. WorkingOwners are different from general Owners in that working owners get tips. An example of a WorkingOwner would be a hot dog stand owner.

You will implement just one method:

- a constructor WorkingOwner() **[4 points]** – sets the name (but note that WorkingOwner inherits its name field) as well as the targetTipPct field.

We are now ready for our last three classes which are shown in these UML class diagrams:



Check [4 points]

Check models a customer's payment. It is used to specify the menu price, sales tax, and tip. You will implement these methods:

- a constructor **[2 points]** – sets the field values; in particular, the sales tax is 15% of the menu price;
- setTipByPct() **[2 points]** – returns nothing (void); This setter method has one parameter which is the tip percentage; it sets the tip to be that percentage of the menuPrice. Note that percentage is a value between 0 and 100.

Customer [12 points]

You will implement two methods:

- a constructor Customer() **[4 points]** – there are two parameters: the name (this is theonly place where the customer identity is used) and the targetTipPct (this is the tip percentage that this customer give for their expected level of service; the tip is determined by the Customer in its dineAndPayCheck method – see next)

- dineAndPayCheck() **[8 points]** – returns nothing (void); the details of this method are given below.

The idea of the dineAndPayCheck() method is that a customer visits a FoodPlace, eats, and pays the check. The tip calculation and the way the payment is distributed depends on the FoodPlace. Specifically, the method does the following:

1. **construct** a new check;

2. **calculate the tip:** The tip percentage is the average of a target tip for that customer and the FoodPlace's tip percentage returned by the getTipPercentage() method of the FoodPlace. For example, a customer might have a target tip of 10% and the food place might have a target tip of 20%, and in this case the customer would tip the average, namely 15%. Another example is a customer with 15% tip percentage who goes to a FastFood place like McDonalds; the FastFood tip percentage is 0 % and so the customer would only give 7.5% tip percentage on the check. These target tips are fields within the Customer and Foodplace objects. Hint: the method Check.setTipByPct() should be called here.

3. **distribute earnings of this check:** the owner gets an income contribution of the menu price; the tip is distributed according to the FoodPlace's class policy; the sales tax is set aside for the tax collector. This procedure is done with the **FoodPlace.distributeEarnings()** method. **[update: Should be FoodPlace.distributeIncomeAndSalesTax() method]**

There is a helper method included in the starter code getDescriptiveMessage() which prints out that a customer went to a particular FoodPlace. You can use it for debugging purposes, if you wish. You can also add a toString() method if you wish; it can help you with debugging.

TaxCollector [8 points]

You will implement the following methods:

- a constructor TaxCollector **[4 points]** - assigns the food place list; you can use the reference i.e. you don't need to make a shallow copy.

- collectTax() **[4 points]**– returns nothing (void); for each FoodPlace in the list of all FoodPlace's, the sales tax is collected and the income taxes from all IncomeTaxPayer's are collected, and these values are added to the incomeTaxCollected and salesTaxCollected fields.

ASIDE: This class is meant to be instantiated once, but we won't require you enforce this in the code). In terms of object oriented design (COMP 303), we would say it has a Singleton design pattern.

2 Where to start? How to proceed?

We suggest that you implement and test in the following order:

1. all the constructors for all the classes and abstract classes – the constructors are relatively straightforward; implementing them will make you familiar with fields in each class;
2. Check
3. IncomeTaxPayer and its subclasses
4. FoodPlace and its subclasses
5. Customer
6. TaxCollector

The reason for doing IncomeTaxPayer before FoodPlace is that the IncomeTaxPayer classes are less dependent on the FoodPlace classes than vice-versa.

3 Submission

Please follow the instructions on Ed Lessons Assignment 2.

- Ed does not want you to have a package name. You can avoid having package name by developing and testing your code in your project's default package. You should be able to drag your files from this package into Ed.
- We will check that your file imports only what you need, namely List, and LinkedList or ArrayList (or both). If you use other imports, your code will not pass this check and you will not be able to submit.
- When you submit, you will see the results of the tests (Test.java). [Updated Oct. 18: see p. 1] ~~Unlike in Assignment 1, we will not give you you hidden tests on this assignment. Rather, we will have private tests. This is much more representative of a real situation, and also more representative of what is done in other COMP courses.~~ Again, we strongly encourage you to come up with challenging tests and to share them with each other.

  If you pass all the exposed tests, then you will get a grade of at least 60/100.
- You may submit multiple times. Your assignment will graded using your most recent submission.
- As on A1, if you submit code that does not compile, you will automatically receive a grade of 0. Since we grade only your latest submission, you must ensure that your latest submission compiles!
- The deadline is midnight Friday, Oct. 29. On Ed, this deadline is coded as 12:00 AM on Saturday Oct. 30. (The reason is that Ed will mark anything submitted at 11:59 PM as late.) Similarly, on Ed, the 2 day window for late submission ends at 12:00 AM on Monday Nov. 1.

Good luck and happy coding!