Assignment 7

Refer to Canvas for assignment due dates for your section.

Objectives:

- Collaborate with others using a shared repository.
- Write generic classes.
- Apply generic classes to variations of a problem.
- Continue to meet the objectives of previous assignments, where applicable.

General Requirements

Create a new Gradle project for this assignment in your ***group*** GitHub repo.

For this assignment, you only need a single package, which you can name as you see fit. The requirements for repository contents are the same as in previous assignments.

GitHub and Branches

Each individual group member should create their own branch while working on this assignment. Only merge to the master branch when you're confident that your code is working well. You may submit pull requests and merge to master as often as you like. Guidelines on working with branches are available from the assignment page in Canvas.

As this is the first group assignment, it is recommended that you practice merging with your group members early on, before you get too deep into the assignment. Getting used to collaborating with GitHub can be painful so don't underestimate this part of the assignment! Two important tips: pull from master before creating a branch and avoid editing files/code that other group members are also working on.

To submit your work, push it to GitHub and create a release on your master branch. Only one person needs to create the release.

The problem

Your team has been commissioned to build part of the back end for a new property company website, similar to Redfin or Zillow. The key entities in the system are properties, contracts (sale vs. rental), listings, and real estate agents.

The company lists the following types of **property**:

- **Residential**
- **Commercial**

All properties need to store the following information:

- **Address**, a String.
- **Size** in square feet, a non-negative integer.

In addition to the above information, **residential properties** also need to include:

- The **number of bedrooms**, a non-negative integer.
- The **number of bathrooms**, a non-negative double (because half-baths are a thing).

Commercial properties also need to include:

- The **number of offices**, a non-negative integer.
- A boolean flag indicating whether or not the property is **suitable for retail.**

Properties (all types) can be listed with the company under one of the following **contract** types:

- **Sale**
- **Rental**

All contracts have:

- An **asking price**, a non-negative double.
- A boolean flag indicating whether or not the price is **negotiable.**

In addition to the above information, **rental** contracts also have a **term** in months, an integer greater than 0, which indicates the length of the contract.

Properties are added to the company website as **listings**. A listing is made up of the **property** and the **contract**.

Agents are people responsible for adding listings to the company site. Agents have:

- A **name**, a String.
- A **collection of their current listings**. You may use a built-in collection type to store agent listings, or you can write your own.
- A **commission** rate, represented as a double between 0 and 1 (inclusive). This is the percentage of the contract amount that the agent takes as payment if they successfully sell / rent the property in one of their listings.
- A **total earnings** amount, represented as a non-negative double. This is the total amount the Agent has earned from their sales / rentals.

Some agents take on any type of listing—residential or commercial, sale or rental—while others prefer to specialize e.g., residential properties only, or sale contracts only. Your design will need to take this into account. For example, it should not be possible to add a listing with a residential commercial property to the current listings of an agent that only handles residential properties.

You may choose to design classes representing properties and contracts as you see fit, but your program must have a `Listing` class and an `Agent` class. ***Both classes `Listing` and `Agent` must be generic***. The purpose of making these classes generic is to make it possible

for an Agent to specialize in a particular type of property or a particular type of contract (or both, or neither), meaning that only appropriate listings can be added to their collection.

Your `Agent` class must include the following methods:

- `void addListing(Listing)`: Adds an (appropriate) listing to the Agent's current listing collection.
- `void completeListing(Listing)`: This method will be called when an Agent successfully makes a sale / rental of one of their listings. Assuming the listing is part of their collection, this method should remove the Listing from their collection and add their commission earnings to their total earnings amount. Commission is calculated from the listing's contract as follows:
    - Sales: The Agent's commission rate multiplied by the asking price.
    - Rentals: The Agent's commission rate multiplied by the asking price multiplied by the term of the contract. For example, if an Agent with a 0.03 commission rate completes a rental that is $2000 for 12 months, their commission would be 0.03 * 2000 * 12 = $720.

  You may decide how to handle the case where the listing passed to method is not present in the Agent's collection.
- `void dropListing(Listing)`: Drop a listing from the Agent's collection without adjusting their earnings. You may decide how to handle the case where the listing passed to method is not present in the Agent's collection.
- `double getTotalPortfolioValue()`: This returns the amount of money the Agent would make if they successfully completed all listings in their collection. To calculate the value of a listing, use the same formulas given for a completed listing.

As always, make sure all your code is thoroughly tested.

Preparing for codewalk

Codewalk is all about explaining your design decisions. In particular, we will want to hear about:

- How you split up the work between team members.
- How your design is *extensible*.
- Where you chose to use inheritance vs. composition and vice versa.
- How you approached the "generic" requirement.
- Any other key design decisions e.g., if you thought a particular class should be immutable, why?