Homework 06 – Monopoly

Authors: Madelyn, Andrew, Sumit, Nicolas, Kevin, Cindy, Jack

Problem Description

Monopoly is a popular board game where players compete to make money and bankrupt their opponents. Each turn players roll a pair of dice to determine how many squares on the board to move. Each square represents a different property (various avenues, railways, etc.). When a player lands on an unowned property, they have the opportunity to purchase it (assuming they have sufficient funds). Purchasing a property means that anytime another player lands on that property, they must pay rent to the owner. Once a player owns a property, they may place houses on it to increase the amount another player would owe should they have the misfortune of landing there. For this homework we have provided you with a working monopoly game.

Despite the game having an element of randomness (provided by the dice), there is a skill aspect to the game. Different players play with different strategies. Your task for this homework will be to teach the monopoly players, represented by the `Player` class, strategies to help them win.

Solution Description

We have provided 6 classes to you that fully implement Monopoly functionality. You are not responsible for creating any classes; you are only responsible for modifying one method inside of `PlayerLoader.java`. We have also included a writeup of the provided files, which can be found in the appendix.

You are responsible for returning an `ArrayList` of `Player` objects in the `loadPlayers` method. A `Player` object takes in a `String`, which represents that player's token, and a `GameStrategy` object. The token shows where the player is on the board. Traditional tokens include a car, a thimble, and a battleship. The `GameStategy` object will determine what the player does in a few different situations. More on this later. Note that `GameStategy` is a functional interface. This means that in order to pass an instance of this interface to a `Player`'s constructor, we must first implement it with a concrete class. In lecture, we have learned a variety of ways to do this. We can implement it with an inner class, an anonymous inner class, or with a lambda expression. Note that each of these approaches fundamentally do the same thing. They are all defining a new class that implements the given interface. Then after defining this class, they create an instance of it. In all cases this instance is an `instanceof GameStrategy`. The only difference between these three approaches is how much syntax you have to write.

In the paragraph above, we noted that the `GameStrategy` instance is used to define how a player behaves after two kinds of events represented as `GameEvents`. The first `GameEvent` is if you land on a property that is unowned and you have the opportunity to purchase it. The second `GameEvent` is when you as a player have the opportunity to buy or sell houses on a property you already own. This object's `handleEvent` method will be called whenever a relevant situation (landing on an unowned property or having the chance to manage the properties you own) occurs. `handleEvent` is passed a reference to the player who will be making the purchase or buying/selling houses, a reference to a `GameEvent` that specifies which of the two situations have occurred, and a reference to the `Property` that the passed in player has landed on.

Note: Feel free to modify Monopoly.java to test out the code you wrote

`PlayerLoader.java`

- Inner Class
  - You should create an Inner Class to be used by the 3rd Player object. More details for the functionality and members of that class can be found in the static method description.
- Method
  - `loadPlayers` – a method that returns an `ArrayList` of `Player` objects. **This is the only method you should be modifying, and you should implement it as follows:**
    - This method is responsible for creating an ArrayList, adding **3 Player objects** to this ArrayList, and then returning that ArrayList. Here is how you should create those Player Objects:
    - The first Player object should have the "Wheelbarrow" token and represents our newbie player. You should create a `GameStrategy` object to initialize it with using a **lambda expression**.
      - The lambda should implement the following behavior:
        - If the GameEvent is `BUY_PROPERTY`, then unconditionally attempt to buy the Property. See the `Player` class for a method we can use to accomplish this.
        - You do not have to explicitly handle the case where we cannot afford the Property – the relevant method in the Player class takes care of this logic for you.
        - For the other GameEvent, simply do nothing.
    - The second Player object should have the "Racecar" token and represents our intermediate player. You should create a `GameStrategy` object to initialize it with using an **anonymous inner class**.
      - The anonymous inner class should implement the following behavior:
        - If the `GameEvent` is `BUY_PROPERTY`, then buy the Property **only if** the current player has at least $500. See the `Player` class for a method we can use to accomplish this.
        - If the GameEvent is `MANAGE_HOUSES`, then you also have 2 cases:
          - If the current Player has at least $1000, iterate through their Property list **from the end to the beginning** and buy 1 house from each Property until you reach the beginning of the list **or** your Player has less than $500.
          - If the current Player has less than $500, iterate through their Property list **from the end to the beginning** and sell 1 house from each Property until you reach the beginning of the list **or** your Player has at least $500.
          - If the Player's money does not meet either condition, then do nothing.
          - In both cases, you do not have to worry if we are unable to buy/sell 1 house on a particular property – the provided method in the Property class you should use already takes care of this logic. Simply pass the relevant value to the Property method and your implementation will be correct.
    - The third Player object should have the "Thimble" token and represents our wildcard player. You should create this player's GameStrategy object by writing an inner class inside of the PlayerLoader class. The inner class should be created as follows:

- It should have an `int` variable named `counter`, as well as a private no-argument constructor that sets `counter` to 0.
- It should implement the `handleEvent` method as follows:
    - `counter` should be incremented.
    - If counter is not yet 20, we are in the early-game strategy.
        - If the `GameEvent` is `BUY_PROPERTY`, then unconditionally attempt to buy the Property **with probability 50%, determined by Math.random()**. See the `Player` class for a method we can use to accomplish this.
        - You do not have to explicitly handle the case where we cannot afford the Property – the relevant method in the Player class takes care of this logic for you.
        - For the other GameEvent, simply do nothing.
    - If `counter` is at least 20, we are entering the late-game strategy.
        - If the `GameEvent` is `MANAGE_HOUSES`, then, if the Player has less than $500, pick one Property randomly and then try to sell 5 houses on that property. If the player has at least $1000, pick one Property randomly and then try to buy 5 houses on that property.
        - If the Player's money does not meet either condition, then do nothing.
        - In both cases, you do not have to worry if we are unable to buy/sell 5 houses on a particular property – the provided method in the Property class you should use already takes care of this logic. Simply pass the relevant value to the Property method and your implementation will be correct.
        - The random index in both cases should be determined **using Math.random()**. For the sake of autogradability, please use Math.random() in the simplest way possible.
        - For the other GameEvent, simply do nothing.

The Checkstyle cap for this homework assignment is 45. Up to 45 points can be lost from Checkstyle errors.

There are 4 checkstyle errors that will appear on provided files when you run Checkstyle locally. The autograder knows to ignore these, so disregard them. Once again, you should not modify any file other than PlayerLoader.java and only submit that file as well.

We reserve the right to adjust the rubric, but this is typically only done for correcting mistakes.

Allowed Imports

To prevent trivialization of the assignment, no imports are allowed other than the import used in the provided files, which consists only of `java.util.ArrayList`

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`

Collaboration

Collaboration Statement

To ensure that you acknowledge a collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one .java file that you submit**. That collaboration statement should say either:

I worked on the homework assignment alone, using only course materials.

or

In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].

Allowed Collaboration

When completing homeworks for CS1331 you may talk with other students about:

- What general strategies or algorithms you used to solve problems in the homeworks
- Parts of the homework you are unsure of and need more explanation
- Online resources that helped you find a solution
- Key course concepts and Java language features used in your solution

You may **not** discuss, show, or share by other means the specifics of your code, including screenshots, file sharing, or showing someone else the code on your computer, or use code shared by others.

Examples of approved/disapproved collaboration:

- **approved**: "Hey, I'm really confused on how we are supposed to implement this part of the homework. What strategies/resources did you use to solve it?"
- **disapproved**: "Hey, it's 10:40 on Thursday... Can I see your code? I won't copy it directly I promise"

In addition to the above rules, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due.

Turn-In Procedure

Submission

To submit, upload the file listed below to the corresponding assignment on Gradescope:

- **`PlayerLoader.java`**

Make sure you see the message stating "HW06 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Run Checkstyle on your code to avoid losing points
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications

Appendix

`Player.java`

This file contains the `Player` class which creates Player objects which will model the players of the game. You need not modify anything in this class, but you should still read all information here:

- Variables
  - `ownedProperties` - an `ArrayList` of `Property` objects representing all `Property` objects the current Player owns
  - `strategy` – an implementation of the `GameStrategy` functional interface from which the handleEvent method will be invoked whenever that event occurs on the current Player's turn
  - `token` – a `String` that holds the type of token one would find in the game (such as racecar, iron, or battleship)
  - `money` – an `int` that holds how much money that player currently has
  - `currentLocation` – an `int` that keeps track of which property that player is currently on
  - `jailTurnCounter` – an `int` that keeps track of how many turns the player has left in jail. If it is 0, the player is not in jail. If it is [1,3], the player is in jail.
  - `playing` – a `boolean` that represents whether the current `Player` is still in the game. When `false`, their turn is skipped.
- Constructors
  - A 2-arg constructor that takes in values for `token` and `strategy` and initializes those variables accordingly. It initializes `ownedProperties` to an empty `ArrayList`, `money` to 1500, `currentLocation` to 0, and `jailTurnCounter` to 0, and `playing` to `true`. **You should be using this constructor in your solution**.
- Methods
  - A `handleEvent` method that takes in a `GameEvent` and `Property` and simply invokes the `handleEvent` method on `strategy`, passing in the current `Player` as an additional parameter. This method is automatically called by the `Game`, you should never be calling it.
  - A `buyProperty` method that takes in a `Property` object and purchases the `Property` object if the `Player` can afford it. **This method is not automatically called anywhere, you may want to use it.**
  - A `payRent` method that takes in a `Property` object and pays the amount owed to the `owner` of that `Property` object. This method is automatically called by the `Game`, you should never be calling it.
  - A `changeMoney` method that takes in an `int` amount and adds that amount to `money`. This method is automatically called by the `Game`, you should never be calling it.
  - A `vibeCheck` method that removes the current `Player` from the game if their money is less than or equal to `0`, also resetting any of their owned `Property` objects. This method is automatically called by the `Game`, you should never be calling it.
  - A getter for `properties` called `getProperties()`, a getter for `playing` called `stillPlaying()`, and a getter for `money` called `getMoney()`. **Feel free to use any of these methods.**
  - Properly implemented `toString` and `equals` methods. **Feel free to use both methods.**

`Property.java`

This file contains the `Property` class which creates Property objects which a Player can add or remove houses. The logic for buying a property is handled in the Player class.

- Variables
  - `name` – a `String` that holds the name of this `Property` object
  - `rentCost` – an `int` that holds the **base value** for the rent amount a Player would have to pay the `owner` for landing on this property.
  - `propertyCost` – an `int` that holds how much a player needs to buy the property
  - `numHouses` – an `int` representing how many houses are on the current property. The effective rent value for this property will increase with each new house.
  - `owner` – a `Player` that can collect rent money when other Players land on the property
- Constructors
  - A 3-arg constructor that takes in values for `name`, `rentCost`, and `propertyCost` and initializes those variables accordingly. It initializes `owner` to `null`, and `numHouses` to 0. This constructor is automatically invoked by the Game, you should never be calling it.
- Methods
  - A method called `changeHouses` that takes in an integer. If the integer is positive, then that number of houses will be added to the `Property` up to a maximum of 5 houses. The `Property` owner will be charged unconditionally for purchasing those houses, even if it makes them lose the game. They pay half of the property cost for each new house. If the integer is negative, then that number of houses will be removed from the current Property object until there are no houses left to remove. The `Property` owner will receive a quarter of the property value for each house sold. **This method is not automatically called anywhere; you may want to use it later.**
  - A method called `getEffectiveRent` that returns the rent a Player would have to pay to the Property owner when they land on this property. The effective rent is `(1 + numHouses) x rentCost`. **This method is used already, but you may also want to use it later**.
  - A method called `reset` that takes in a `Player` object. If the `Player` object is no longer in the game, the state of this `Property` is reset. This method is automatically called, you should never be calling it.
  - A getter method for `propertyCost` called `getCost()`, and a getter for `owner` called `getOwner()`. **Feel free to use both methods.**
  - A conditional setter for `owner`. This method is automatically called, you should never be calling it.
  - Properly implemented `toString` and `equals` methods. **Feel free to use both methods.**

`GameStrategy.java`

This file contains the `GameStrategy` functional interface which has one method to perform actions in the game.

- Method
  - a `handleEvent` abstract method that takes in a `Player` object, a `GameEvent` enum, and a `Property` object, in that order.
    - When implemented, `handleEvent` should handle each possible `GameEvent` separately. Note that the `Property` object will be `null` in the `MANAGE_HOUSES` state – the purpose of this state is for the Player to manage *all* their properties, not just one.
    - **Since every Player requires a GameStrategy, you should be implementing this method every time you create a Player**.

`GameEvent.java`

This file contains the GameEvent enumeration that represents a state during the Game

- Enum values
  - `BUY_PROPERTY`: Represents the point in the Game where a Player has landed on an unbought Property and *may or may not* choose to buy the Property.
  - `MANAGE_HOUSES`: Represents the point in the Game which occurs at the end of every Player's turn; it enables them to add or remove houses to any of their owned Property objects as desired.
  - **You should be using one or both of these values every time you implement the handleEvent GameStrategy method**

`Monopoly.java`

This file is given to you and contains the `Monopoly` class which contains a handful of static methods and variables that progress the game. It also, notably, contains a main method that you can use for testing.

- Static Variables
  - `PROPERTIES` – a constant array of `Property` objects such that each index represents a location on the board. Note that every game space is a property space either unowned or owned by an active Player.
  - `playerList` – an `ArrayList` of `Player` objects representing all Players who started the game, even if some of them are now eliminated. This variable is automatically modified by the Game, please do not modify it.
  - `playerTurn` – an `int` representing the index in `playerList` pointing to the `Player` whose turn it is. This variable is automatically modified by the Game, please do not modify it.
- Static Methods
  - `play` – a method that controls all Game logic, running the Game to completion based on the loaded Players. Do not modify this method.
  - A `main` method that, by default, populates `playerList` with the result of `PlayerLoader.loadPlayers()` and then calls `play()`. **Feel free to modify this main method if you wish to do any additional testing.**