SYSC 2004
Winter 2018
Lab 11

Objectives

1. Event Handling
2. Observer Pattern
3. Exception Hierarchy
4. Practice, Practice, Practice

Provided: Solution for Lab 8 as well as Board.java (to save you time)

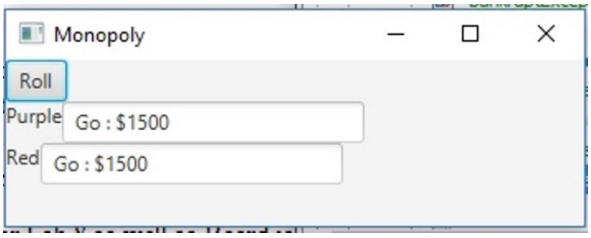
Submission Requirements: (Exact names are required by the submit program)

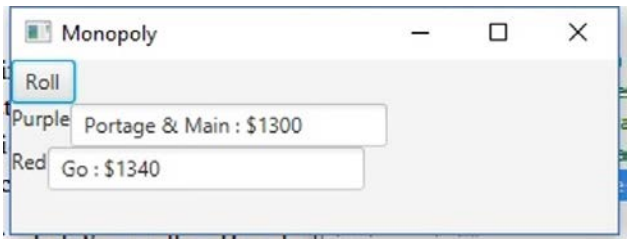
- **Without a submission you will not get a mark**
  MonopolyGame.java, BankruptException.java, IncomeTaxSquare.java, Player.java, PropertySquare.java

Background

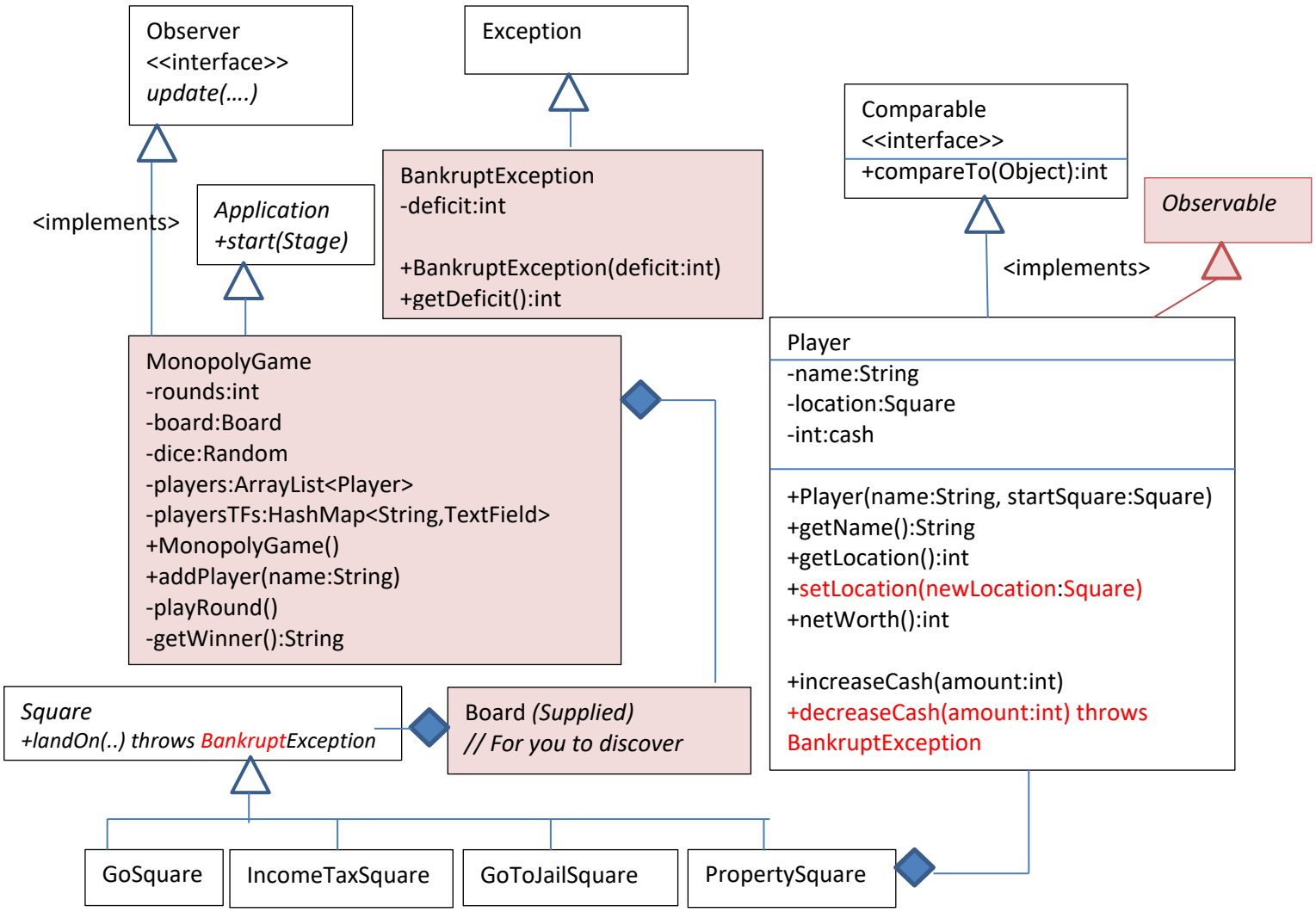
We are going back to the Monopoly game of Lab 8 to build a simple game engine and a very simple GUI. You are free to try out a full grid of squares, but the GUI suggested below is a quick method using elements that you know already. You are also free to add more than two players.

- Whenever the **Roll** button is clicked, another round of the game is played by the computer. In a single round, all players take a turn: The dice is rolled, the player's location is updated appropriately so that they `landOn` a new square (and any appropriate actions are taken).
- Should a player's cash be reduced to/below zero, the player is out of the game.
- The game is limited to 10 rounds. After that, an ALERT.INFORMATION dialog is popped up and the program terminated.
- Limitations: Once you get into Jail, you don't get out





UML Class Diagram (Changes are highlighted in colour)



Part 1 : Preparing the project

Make a new project called **lab11** that is a copy of **lab8** (or the posted solution).

Part 2 : A Specialized Exception

1. Add a new Java class to the project, called `BankruptException`.
2. Implement the `BankruptException` class as shown in the UML. It is a simple exercise, but it is significant in that you are creating a new kind of application-specific exception that contains data. Remember the process.
3. Make use of this new `BankruptException`.
    - `Player::decreaseCash`(…) should now advertise and throw a `BankruptException` whenever the amount of cash goes below zero.
        - When you make this change, the compiler will now raise errors in other classes … those that call the method `decreaseCash`(). You have to fix these compiler errors.
        - Example: The `IncomeTaxSquare::landOn`() method calls `decreaseCash`() so it must now advertise `BankruptException` (*or internally catch it*).
    - Depending on your particular solution: If/Where you have thrown a generic `Exception` to flag a player who runs out of money, change that code to throw a specific `BankruptException` (including the amount of money that player owes)
        - Example: `PropertySquare::landOn`() should advertise `BankruptException`
4. Question: What code will catch the exception? `MonopolyGame.playRound`(), but we're get to that in the next part.

Part 3 : The Observer Pattern

1. No coding, just think. The two participants of the Observer Pattern are identified for you in the UML class diagram. What is left for you to figure out is what "state" is being observed. In what method of the `Observable` does the state change occur (and where its observers should be notified)?
2. If you know the answer, make the appropriate changes to your code. If you don't know the answer yet, it's okay. Carry on, with Part 3.

Part 4 : The MonopolyGame

The `MonopolyGame` class is both a game engine and a GUI. A game engine keeps track of the players and advances the players in turn. In this game, a game engine amounts to repeatedly rolling a simulated dice and advancing the next player's location.

1. No coding, just think. Study the UML class diagram for `MonopolyGame`, and figure out which variables and methods concern the game engine, and which elements concern the JavaFX application. This will help you manage the next few steps.

2. Add a new **JavaFX Application Class** to your project, called `MonopolyGame`. For now, ignore all the JavaFX stuff, and consider it simply as a common class.
    - Declare all the required instance variables, as shown in the UML.
3. Implement the elements of `MonopolyGame` class that concern the game engine.
    - Implement the default constructor. Constructors initialize all the instance variables.
        - Initialize `rounds` to 10, the maximum number of rounds in our simple game. It should be decremented as each round is played. When it reaches zero, the game should end.
        - Initialize an empty `players` list. Players will be added to this list as they join the game.
        - Initialize an empty `playersTF` map. More on this later, when we consider the GUI aspects of this class. For now, it should be a valid empty map.
    - Implement the `addPlayer`(..) method
        - The argument to this method is the desired `name` of the new `player`. It should be used as part of the instantiation of a new `player` object, located on the GO `square`.
        - The `player` object should be added to the list of `players`.
    - Implement the `playRound`() method
        - Notice the visibility modifier for this method.
        - A round of a game lets all players take a turn.
        - The method should iterate through the list of `players`. One-by-one, the `dice` should be rolled, the new location of that player calculated, and the player should then `landOn` the new location.
            - **Hint**: There is a handy method in the `Board` class for figuring out the new location (that handles going past GO).
            - **Caution**: The `dice` should roll between 1 and 12. Random returns a number between 0 and max. You have a mapping problem to solve.
        - The method must terminate the game after 10 `rounds` of play.
        - The method must also terminate the game if a player goes bankrupt. (In the real game of Monopoly, play would continue, but in our game, the game ends when any single player goes bankrupt). Terminate your program with the following statement:

```
System.exit(0);
```

    - Implement the `getWinner`() method
        - Notice the visibility modifier for this method.
        - I wonder how you should compare these players to find the winner??
4. Now, it is time to implement the elements of `MonopolyGame` class that concern the Graphical User Interface. As a JavaFX `Application`, most of the remaining code belongs in the `start`(..) method.
    - The first thing that the `start`(..) method will do has nothing to do with the view. It will add a few players to the game. You are free to add whatever players you like,

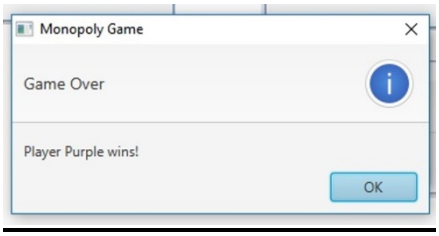
but you must have at least two. In the sample program, there are two players, called *purple* and *red*. At this point, your `players` list should now be non-empty.

- Populate your view. By this time, you should be able to do this by yourself, with only one tidbit of explanation about the `playersTFs` hashmap. You are going to be creating a `TextField` object for each player in the `players` list. This same `TextField` object will need to be updated as the game is played, with the associated `player's` new location. Consequently, you need to retain the reference of the `TextField` object. Moreover, you need to be able to find the `TextField` object belonging to a given player. This is the role of the `playersTFs` hashmap. Within this `start`(…) method, as you create a `TextField`, store the <`player's name, TextField object`> in the hashmap, for later use.
- Complete this phase of development by creating a minimal event handler for the one button.
    - You are free to write the event handler as a named inner class or an anonymous inner class (or even a lambda expression!). It must be an inner class, though; not a non-public class. This class is not shown on the given UML
    - The `handle`() method should simply print out "Roll", for now.
- Test your program. Finesse the look of your window, perhaps changing the size of your scene object because some of the property names are pretty long. And, verify that clicking on the Roll `button` results in a console message being printed. If not, have you forgotten to register your handler on the button itself (i.e. call `setOnAction`(..))?

5. Complete the behaviour of the GUI so that each player's location and net worth are updated in their respective `textfield`, as each round is played. To do so, you need to complete the implementation of the Observer Pattern.
    - If you have not already done so, in the `Observable` (i.e. the Player) you must call `setChange`() and `notifyObservers`() "somewhere".
        - Solution: In `Player`'s `setLocation`() method because the `textfield` should be updated whenever a `player`'s location is changed.
    - In the `Observer`, you must:
        - Add `this observer` to (each) `observable` object.
        - Implement the `update`(…) method.
            - Downcast the `Observable` argument to a `Player` object.
            - Get the `name` of the `Player` object
            - Get the `textfield` object associated with that `player's name`, from the hashmap.
            - Update that `textfield` object with a string that is composed of the `player`'s current location and net worth.
6. Now it is time to run your game. You should see the players' locations and net worths change. After 10 rounds, the program should quit.

[Optional] Part 5 : Learning about JavaFX Alerts

Right now, your program abruptly ends when either 10 rounds have elapsed or a player has gone bankrupt. It would be so much nicer if we displayed an informative dialog



The pictures shows a JavaFX Alert, in particular as JavaFX Information Alert dialog. A dialog is an independent window that pops up and is dismissed, before the main application continues on. Some good examples are available at this URL: https://o7planning.org/en/11529/javafx-alert-dialogs-tutorial

Step-by-Step Instructions:

1. In the `playRound`() method, there should be two places where you terminate your program using `System.exit`(0): when you catch a `BankruptException` when a player `landsOn` a new square; and when the number of `rounds` has exceeded ten.
2. Insert the following code before your calls to `System.exit`(0);

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Monopoly Game");
alert.setHeaderText("Game Over");
alert.setContentText("Player " + getWinner() + " wins!");
 alert.showAndWait();
```

For the case when a player has gone bankrupt, change the ContentText to also include the name of the player who went bankrupt.

Enjoy!