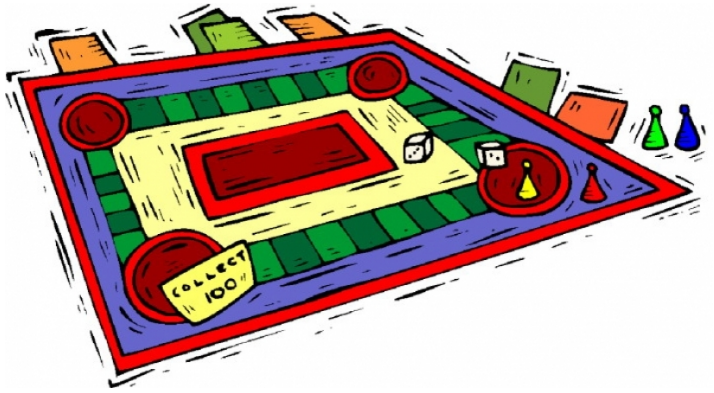
COMP 1040 Programming Fundamentals
SP2 2018
Assignment 2

Stephen Searle

May 6, 2018



"I think it's wrong that only one company makes the game Monopoly."
- Steven Wright.

1 The Problem

The supplied code, when run, simulates a game of Javopoly™. However the game is very boring: all of the squares on the Javopoly board are empty so nothing happens when a player lands on any of them. Your task is to make the simulation a little bit more interesting by implementing the following:

1. Implement a class hierarchy for all the types of squares on the board.
2. Implement a class hierarchy for the various types of LuckyCards which are used when a player lands on a PotLuck square.
3. Fill in the methods of the OutputPlayers, which write information about `Players` in the game to a text file.
4. Create an Exception class and modify the GameBoard class to throw this Exception when too many squares of any type are added to the board.

2 Description of Classes

2.1 game.JavopolyGame

JavopolyGame is the main class; the `main` method sets up and runs the simulation. The Javopoly game has a GameBoard, a pile of LuckyCards (known as a CardStock), and an array of Players. Once all the components are set up the game cycles through each Player in turn, rolling the dice, moving the Player, and handling what happens when the Player lands on each BoardSquare. Players can go into debt (have a negative cash balance). In this case a Player is prevented from moving until his/her balance returns to positive (*e.g.* through collecting rent). Each time a Player completes a circuit of the board they are awarded $200 from the bank. Prior to each dice roll the user is given the option to roll (continue the game), dump (save Player data to text file) or quit.

The following methods are all visible and may be used by your own classes:

- `int getNumPlayers()`
- `Player getPlayer(int)`
- `GameBoard getBoard()`
- `int rollTheDice()`
- `int getLastDiceRoll()`
- `LuckyCard getLuckyCard()`
- `boolean movePlayer(Player, int)`

The JavopolyGame supplied to you has been limited in order to run as-is without any extra classes. When you have written and added your own classes you will want to make some changes:

1. in method `setupCards` comment out the first line and uncomment the second line, once you have implemented all of the LuckyCard types. Also edit the CardStock constructor to enable method 2.
2. in constructor `JavopolyGame` comment the `testInitialisation` line then uncomment one of the other lines (or add a call to your own initialisation routine) to set up a more interesting game. Also edit GameBoard to uncomment/enable the other initialisation routines.

2.2 game.Player

This class provides functionality for a single player in the game. Each Player has a cash balance, a position on a GameBoard, and a portfolio of Propertys. At most `Player.MAX_NUM_PROPERTIES` can be owned by a single Player.

You should not need to make changes to this class, but you will use its publically visible methods in your own classes:

- `double getCash()`
- `void addToCash(double)`
- `void decreaseCash(double)`
- `int getSquareIndex()`
- `int advancePosition(int)`
- `int addToPortfolio(Property)`
- `getNumProperties()`
- `getNumPropertiesOfSameType(Property)`

See header comments / JavaDoc to learn how to use these methods. In addition to the above there are several methods which return information about the Player as a String: `playerID()`, `playerPosition()`, `listOfProperties()` and of course `toString()`.

Be aware that each Player receives a unique colour. Once these are exhausted it will not be possible to create new Players. If you require more Players then edit the class and add more colours to `enum COLOUR`.

2.3 game.GameBoard

The GameBoard maintains an array of all the squares on a Javopoly board and provides functionality to set up the board. Use the method `getSquare(int)` to obtain any BoardSquare by its index. The method `setSquare(int, BoardSquare)` is used to affix a BoardSquare to a particular index on the board.

The supplied GameBoard class will allow you to set any number of squares of a given type. However, the defined constants `MAX_NUM_BUSINESS, MAX_NUM_BUSSTATION` and `MAX_NUM_GAOL` are all supplied. Your fourth task will involve modifying or overloading `setSquare` so that it throws an Exception if a user tries to set a square to a Business, BusStation or Gaol when the respective limit of that square type has already been reached.

The supplied GameBoard will only create a board full of default squares (which do nothing). Once you have implemented all required classes then uncomment the `mawsonLakesInitialisation` method and also the `case` blocks in `testInitialisation` so you can create a board with more interesting square types. You can also add your own initialisation code here (and call it from the JavopolyGame constructor) to set up boards with whatever squares you require.

2.4 game.CardStock

This class is used by the GameBoard to serve LuckyCards when required. As supplied it creates a stock of DoNothingCards. When you have written classes for all of the LuckyCard types, uncomment `case 2` in the constructor, and make the change to `JavopolyGame.setupCards`. You can also write and use your own `case` to set up a stock of LuckyCards as you desire.

2.5 squares.BoardSquare

This class represents a square on the Javopoly board. It has a name which is supplied as a String upon construction. It has a reference to the JavopolyGame object to which it belongs, also supplied upon construction. BoardSquare methods may call all of the JavopolyGame's visible methods via this reference. BoardSquare has the following methods:

- `boolean isBuyable()`
  return `true` if this square can be bought by a Player.
- `boolean handlePlayerLeaving(Player)`
  Process an attempt by a Player to leave this square. Return `true` if successful, `false` if the Player did not leave the square.
- `void handlePlayerEntry(Player)`
  Process what happens when a Player enters this square.

2.5.1 Kinds of BoardSquare

In addition to a generic BoardSquare are the following types.

PotLuck: When landed upon, a LuckyCard is drawn from the CardStock and applied to the Player.

Gaol: A Player on a Gaol square is permitted to leave the square only on a dice roll of 2,3,11 or 12. Gaol is entered only by landing on it; there is no "Go to Gaol" square.

Property: A square which can be bought and used to collect rent from other Players. See next section (§ 2.5.2).

2.5.2 squares.Property

NOTE: A Property class has been supplied only so that the other supplied code will compile and run. You must replace this with your own Property class which behaves as follows.

A Property is a BoardSquare which can be bought by a Player. It has a price and a name (supplied at construction), and may eventually have an owner. When a Player enters a Property square,

- if (s)he owns it, a message to this effect is displayed.
- if another Player owns it, rent is paid to him/her.
- if
  - it is unowned, and
  - the Player can afford it (*i.e.* (s)he has enough positive cash balance), and
  - the Player does not already own `Player.MAX_NUM_PROPERTIES` Propertys,

  then it is bought by the Player: his/her balance is decremented by the purchase price and it is added to his/her portfolio.

There are three kinds of Property and each has a different way of computing rent amounts:

Street: rent is the purchase price divided by 10. There are no houses or hotels in Javopoly!

BusStation: rent is equal to $50 multiplied by the total number of BusStations in the owner's portfolio.

Business: rent is computed as $10 multiplied by the last dice roll, multiplied by the total number of Businesses in the owner's portfolio.

2.6 cards.LuckyCard

This is the base class for all kinds of `LuckyCard` in the game. All potluck card types must be derived from this, directly or indirectly. Be aware of the following methods:

- `LuckyCard(String)` The base constructor takes a message which is displayed when the card is enacted. *Note that subclass constructors may require extra information, e.g. monetary amounts.*
- `showMessage()` prints the card's message, as supplied in the constructor
- `abstract boolean enactCard(Player, JavopolyGame` Execute the actions of the card upon the supplied Player, in the supplied JavopolyGame. When you override this method in your subclasses you will exploit methods of Player and `JavopolyGame`.
- `String toString()` This is used to print out the required text when the card is executed. It will be overridden by your subclasses.

2.6.1 kinds of LuckyCard

cards.DoNothingCard is supplied for you; when enacted, a DoNothingCard displays "Nothing to do", and does nothing else. Other kinds, which you must implement, are

AdvanceToken: This has a number of steps, which is specified on construction. When enacted on a Player, that player is moved ahead by that number of squares.

PayAllPlayers: A money amount is supplied on construction. When enacted on a Player, that player must pay the specified amount to each other Player in the game.

PayToBank: A money amount is supplied on construction. When enacted on a Player, that player has his/her balance decreased by the specified amount.

ReceiveAllPlayers: A money amount is supplied on construction. When enacted on a Player, that player receives the specified amount from each other Player in the game.

ReceiveFromBank: A money amount is supplied on construction. When enacted on a Player, that player has his/her balance increased by the specified amount.

RollAgain: The dice are rolled again and the Player is moved accordingly.

2.6.2 useful methods of other classes

You may find the following methods of other classes useful when you write the `enact` method for each card type. LuckyCards:

- `JavopolyGame.movePlayer()`
- `JavopolyGame.getNumPlayers()`
- `Player.decreaseCash()`
- `Player.addToCash()`
- `JavopolyGame.getPlayer()`
- `JavopolyGame.rollTheDice()`

2.7 game.OutputPlayers

This class is used to write information about all Players to a text file. Two `writePlayersToFile` methods are supplied. Your third task is to replace the code within these methods in order to open a text file and write Player information to it.

2.8 utility.DiceRoller

This class provides functionality to roll a number of dice having a given number of sides. In Javopoly two six sided dice are used; this is hardwired into the game as a named constant.

When debugging and testing your code you may want to "load the dice" to return predetermined values. The method `DiceRoller.setSeed` allows you to load the dice with a specific sequence of numbers, or even allow the user to input the result of every dice roll. If desired, then edit `JavopolyGame.main` to set the DiceRoller behaviour before the `playTheGame` method is called.

3 Tasks

3.1 Squares of the Board

Design and implement a class hierarchy for all types of squares on the board as described in § 2.5, with BoardSquare as the root class. Implement all such classes in the `squares` package. Use inheritance where appropriate, use abstract where appropriate, implement methods at an appropriate level in the inheritance tree.

Your classes must work with the supplied code, *e.g.* `GameBoard.mawsonLakesInitialisation`. Examine this code and make sure your class names and constructors agree with these calls.

3.2 Lucky Cards

Design and implement a class hierarchy for all PotLuck cards as described in § 2.6, with LuckyCard as the root class. Implement all such classes in the `cards` package. Use inheritance where appropriate, use abstract where appropriate, implement methods at an appropriate level in the inheritance tree.

Your classes must work with the supplied code, *e.g.* `CardStock()` called with `initialise` parameter equal to 2. Examine this code and make sure your class names and constructors agree with this.

3.3 Player output

When "dump" is entered into the simulation prompt, details of each Player should be written to a text file. Edit the OutputPlayers class and implement the two `writePlayersToFile()` methods. These should open the specified text file, write information about each Player in the specified JavopolyGame, and close the file. An error message should be written to the console in case of error.

HINT: use `JavopolyGame.toString()`.

3.4 Limit number of square types

A Javopoly Board should only allow a limited number of squares to be Businesses, BusStations or Gaols. The maximum number of each type is specified in `GameBoard.MAX_NUMBER_BUSINESS`, `.MAX_NUMBER_BUSSTATION` and `.MAX_NUM_GAOL` respectively.

Write a new Exception class called `TooManyException`. Modify the GameBoard class to throw a `TooManyException` if an attempt is made to set a square of the board to one of the limited types, when the maximum allowable squares of that type has already been reached. When throwing the Exception ensure that the message mentions which limit has been violated.

HINT: modify or overload `GameBoard.setSquare()`.

3.5 Testing

game.TestRoutines contains a couple of methods for testing aspects of the game: one for testing the AdvanceToken lucky card, the other for testing the purchase of Streets. Each method returns `true` if the test passes. The `main()` method runs each test method.

Devise more test methods for at least four other aspects of the game. You can choose what to test. Implement your tests in this class. Add calls to them from the `TestRoutines.main()` method.

Comment the headers of your methods to describe what you are testing, how you are doing it, and what you expect the output to be if the test is successful. Even if you don't write the actual methods, write the comments describing the tests and you'll be awarded half marks.

Note: to use `assert` in your test routines you will need to enable assertions in eclipse. See header comments in TestRoutines.

4 Marks breakdown

- **30** design and implementation of class hierarchies (tasks 1 and 2)
- **5** class hierarchy diagram (tasks 1 and 2)
- **15** implementation and correct use of of TooManyException (task 3)
- **10** file output (task 4)
- **10** implementation of testing methods.
  Half marks for comments/plan only.
- **10** reasonable output produced when run with Mawson Lakes board.
- **10** commenting. This must include JavaDoc for each method.
- **10** other style: indentation, meaningful identifiers, etc.

5 Advice

I suggest this plan of attack:

- Tasks 1 and 2:
  - Sketch out a class diagram for the BoardSquare and LuckyCard class hierarchies. Try to identify commonalities among the various squares and cards and reflect this with intermediate level classes.
  - Implement all the classes as stubs, *i.e.* classes with empty method bodies (except maybe for `super()` calls in constructors). For example write all the card classes like DoNothingCards. Ensure that your classes will compile with the supplied code, *i.e.* they have the same names and the same constructor signatures as used in the GameBoard and CardStock initialisation routines.
  - Make the following edits to enable the full game:
    - edit the JavapolyGame constructor, comment out the `testInitialisation` line and uncomment one of the other initialisation lines.
    - edit `setupCards` method, comment the first line and uncomment the second.
    - edit CardStock and uncomment case 2 in the constructor
    - edit GameBoard and uncomment the `mawsonLakesInitialisation`.

    Run the program. This will confirm that you have all the required classes with the correct method signatures.
  - Implement each class in turn. Fill in the methods, test the methods, run the program. When you're satisfied then move to the next class. Consider writing methods in TestRoutines as you go.

- Task 3: Implement methods in OutputPlayers class.
- Task 4: Implement TooManyException and employ it in GameBoard. Initially work with only one of the limits, *e.g.* `MAX_NUM_GAOL`. When you get this working extend your code for the other two limits.
- Testing: add tests (or at least test plans in comments) in TestRoutines, if you have not already done so while developing your classes.
- Export your project as a ZIP file (ensuring class hierarchy diagram is a file in the top level of your project, *NOT* as a separate entry in the ZIP). Submit this.
- Kick back and relax! ☺

General advice:

- Pay attention to the course discussion board. Any announcements or corrections will be made here.
- Seek help if you need it! Ask fellow students on the discussion board, come to consulting sessions with the lecturer, etc.