Lab 1: Silver Dollar Game[1]
CSCI 2101 – Fall 2017

Due: Wednesday, September 13, 11:59 pm
Collaboration Policy: Level 1 (review full policy for details)
Group Policy: Individual

This lab will give you experience writing an object-oriented program in Java as well as working with one of the simplest types of data structure (an array). To do so, you will write a program allowing you to play a simple two-player game, the Silver Dollar Game. The rules of the game are described below, followed by some suggestions for how to go about designing and implementing your program.

1 Game Rules

The Silver Dollar Game is played between two players. An arbitrarily long strip of paper is marked off into squares, as pictured below:

The game begins by placing silver dollars in a few of the squares. Each square holds at most one coin. Interesting games begin with some pairs of coins separated by one or more empty squares. For example:

From the starting board configuration, players take turns moving a single coin, constrained by the following rules:

1. Coins move only to the left.
2. No coin may pass (skip over) another.
3. No square may hold more than one coin.

The game ends once the n coins occupy the leftmost n squares of the board (that is, no further moves are possible). The final player to move is the winner.

[1] Adapted from a lab provided with *Java Structures*, D. Bailey

2 Game Interface

Your program will facilitate playing the Silver Dollar Game using a simple text-based interface. At the beginning of each round, a textual representation of the board should be printed to the terminal. The active player (Player 1 or Player 2) should then be prompted for a move, e.g.:

```
Player 1: Enter your move:
```

A move should be specified as two integers: a coin number (where the leftmost coin is coin 0) and the number of spaces to move that coin. For example, entering `3 2` says to move coin 3 (the fourth coin) two spaces left. Note that in order to facilitate ease of play, your textual board representation should make it clear which coin is which (e.g. coin 3 vs coin 4).

For now, you can make the assumption that the user will only enter numbers when prompted (and not arbitrary text). Gracefully handling malformed input would be nice, but would require using exceptions, which we haven't covered in Java (but you may remember from Python)!

Note that many syntactically-valid moves (two numbers) may still not be valid moves given the current board configuration. For example, the move `5 3` isn't valid if there is no coin 5, or if coin 5 exists but can't be moved left 3 spaces without falling off the edge of the board or violating one of the game rules. If such an invalid move is entered, a message indicating as such should be printed, and the current player should again be prompted for a move, e.g.:

```
Player 1: Enter your move: 5 3
Illegal move!  Try again.
Player 1: Enter your move:
```

Once the game is won, the final board should be printed along with the winner, e.g.:

```
Player 2 wins!
```

3 Program Design

You should write your program in a single Java class called `CoinStrip`.

Before you write any code, you should plan out the design of your program. For programs of any significant complexity, tackling design before beginning to implement is very important. You should first decide on an internal representation of the coin strip – i.e., the instance variables that your class will keep track of and what they represent. Make sure your representation supports all needed operations, such as testing for a legal move, printing the current board, testing for a win, moving coins easily, etc. Think carefully about your representation and consider alternatives – the most obvious representation may not be the best one!

Once you have decided on a representation, write down the set of operations offered by your class. In other words, what are the public methods of `CoinStrip`, what parameters do they take, what do they return, and what do they do? As a simple example, your class might have a public method that checks if a specific move is legal for the current board state.

The `main` method of your class should create a `CoinStrip` object representing the board and then begin prompting the players to make their moves. In other words, the public methods of your `CoinStrip` class will be called from within your program's `main` method, which will actually control the game simulation. Note that your `main` method should *only* interact with the `CoinStrip` object via its public methods (and not by directly accessing its instance variables).

4 Implementation Tips

You have significant freedom in deciding how to implement your program (provided it follows the game rules and interface specifications given above), but below are some suggestions for your implementation.

As with all programs, you should build your program incrementally, compiling and testing as you go. Don't blindly try to write the entire program before testing anything! An incremental approach will both make your debugging more efficient as well as help in producing a modular program design. A good general approach is to write one method, test it to make sure it works as intended, then move onto another method. Aim for small, compact methods that accomplish specific jobs. Once all your methods are working, you can combine them in the `main` method of the final program.

Some more specific pieces of advice are below:

- The simplest data structure to store the game board is an array. You can also use a fancier data structure like an `ArrayList` in roughly the same way, but a basic array should be fine for now. You will still need to consider what the array is actually storing!
- Use the built-in `Scanner` class to handle reading user input, as demonstrated in class. Remember that you will need to include `java.util.Scanner` in order to use the `Scanner` class.
- Use the built-in `Random` class to generate the initial board configuration (you'll need to include `java.util.Random`). Remember that a `Random` object is a random number *generator*, not a random number itself. Random integers can be retrieved from the generator using the `nextInt` method.
- One of your challenges is to generate the initial board configuration. To keep things simple, you can just pick a fixed number of coins to place (e.g., 3 or 4), or you can randomize the number of coins. You will still need to decide how to assign the starting coin positions, however.
- One of the most important functions of your program is generating the textual board representation to print each round. This representation should be returned by your class' `toString` method. If you have a fancy representation, this method may actually be fairly substantial. It's a good idea to start with a very simple representation, such as the one below:

```
_ _ _ 0 _ _ _ _ _ 1 _ _ _ 2
```

  Once that's working (and possibly when the rest of the program is working as well), you can change `toString` to return a fancier representation, such as the one below:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   | 0 |   |   |   |   |   | 1 |   |   |   | 2 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

  Note that while you are not required to produce a nicer representation such as this one, it's a good programming exercise to do so!

5 Thought Questions

Once your program is finished, consider and answer the following questions:

1. How might one pick game sizes (number of coins) such that one has a 50% chance of three coins, a 25% chance of four coins, a 12.5% chance of five coins, and so on?
2. Does your method of generating the starting configuration guarantee that the game is not an immediate win? If not, how might you change your approach to enforce that guarantee (without just adding some blank spaces to the left of an existing configuration)?
3. A similar game, called Welter's Game (after C. P. Welter, who analyzed the game), allows the coins to pass each other. Would this rule modification change your implementation significantly? Why or why not?

Write your answers in the `README.TXT` file that is automatically generated by BlueJ (which you can also open by double-clicking the page icon in the BlueJ window). You should also include your name (but can delete everything else that BlueJ automatically puts in that file).

6 Evaluation

Your completed program will be graded on design, documentation, style, and correctness. Make sure that your program is documented appropriately and is in compliance with the CSCI 2101 Java Style Guide. Lastly, make sure your name is included in both your `CoinStrip.java` file and your `README.TXT` file.

7 Submitting Your Program

Once your program is finished, you should follow the following steps to submit:

1. Save your program and **quit BlueJ** (this is necessary because BlueJ gets confused if you perform step 2 – renaming your project directory – while the project is open).
2. Rename your project folder (which is the folder that contains your `.java` files, `package.bluej`, and possibly a few other files) so that it is named `username-lab1` (with your actual username). For example, I would rename my folder `sbarker-lab1`.
3. Create a single, compressed `.zip` archive of your project folder. On a Mac, right-click (or, if you have no right mouse button, control-click) on your project folder and select "Compress your-folder-name" from the menu that appears. On a Windows machine, right-click on the folder, select "Send To," and then select "Compressed (zipped) Folder." In either case, you should now have a `.zip` file that contains your project, named something like `sbarker-lab1.zip` (with your actual username).
4. Open a web browser and go to the course's Blackboard page, then browse to `Lab Submissions`. Click on `Lab 1` and then `Start New Submission`. In Section 2, you can, but do not need to, provide any comments. Then select `Browse My Computer` and browse to the `.zip` file you created in step 3. Select that file, then click on `Submit`.

After submitting your lab, remember to save a copy of your project folder somewhere other than on the desktop of the machine you are working on (if you're on a lab machine). If you just leave it on the desktop, it will only be available on that machine – if you log into any other machine on campus, it will not be there. You can also store your projects in Dropbox (or any similar service) or in your folder on the `microwave` server (see below).

Connecting to the `microwave` server

All students have storage space available on the `microwave` file server, which you can use to store your labs. The benefit of using this server is that you can access your `microwave` folder from any machine on campus, not just the specific lab machine on which you were last working.

To connect to `microwave` on any lab Mac, click on the Desktop, then select "Connect to Server" under the Go menu (or just command-K). In the window that appears, type the following (substituting your actual username for the word `username`):

```
smb://microwave.bowdoin.edu/home/username
```

Click Connect and enter your Bowdoin password when prompted. A folder should then appear on the desktop named with your username. This is your `microwave` folder, in which you can store your labs when you are finished with them (or leaving but returning to work later). However, you should not work directly on files on microwave. While you are actively writing your program, you should copy your project folder to a "local" area first, such as the desktop. Once you are finished working, you can copy your files back to microwave.