# Lab 1: Silver Dollar Game[1]
## CSCI 2101 – Fall 2021

**Due (Section A):** Monday, September 20, 11:59 pm
**Due (Section B):** Tuesday, September 21, 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 Program Specification

Your program will facilitate playing the Silver Dollar Game using a simple text-based interface, the operation of which is detailed here. At the beginning of the game, the user should first be asked to enter the number of coins:

```
Enter the number of coins:
```

The user should then enter an integer greater than 0 to specify the number of coins (you can assume that the user enters a valid number). The game board containing the desired number of coins is then randomly generated by placing each coin a random number of squares away from the coin to its left (or the left edge of the strip, for the first coin), chosen each time from 1 to 6. Note that this setup procedure means that the starting strip contains at most $k \times 6$ squares for $k$ coins.

Following setup, gameplay begins, starting with Player 1. At the beginning of each move, a textual representation of the board should be printed to the terminal. The board representation should consist of the squares of the strip, separated by spaces, in which each square is denoted by either an underscore (for unoccupied squares) or the number of the coin occupying that square. Coins should be numbered sequentially starting from zero, where coin 0 is the left-most coin. For example, below is a 14-square strip containing three coins (numbered 0, 1, and 2):

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

Following the printing of the board, the active player (Player 1 or Player 2) should then be prompted to enter a move. For example:

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

The player should then specify a move by entering two integers (separated by a space), where the first integer is the number of the coin to move and the second integer is the number of squares to move that coin. For example, entering "`3 2`" says to move coin 3 (the fourth coin) two squares to the left.

Note that many integer pairs may correspond to illegal 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 squares without falling off the edge of the board or violating one of the game rules. If any illegal move is entered, an error message should be printed and the current player should again be prompted for a move. For example:

```
Player 2: Enter your move: 5 3
Illegal move!  Try again.
Player 2: Enter your move:
```

Once the game is won (i.e., no further moves are possible), the ending board configuration should be printed, followed by a victory message for the appropriate player, e.g.:

```
Player 1 wins!
```

You can assume that all user input throughout the game will be in the expected format (i.e., one integer for specifying the number of coins and two integer for each move). Gracefully handling illegally-formatted user input (such as if a user entered "abc") would be nice, but would require using exceptions, which you may remember from Python but we have yet to cover in Java.

## 3 Program Design

You should write your program in two Java classes: a class called `CoinStrip` representing the game board, and a class called `PlayGame` that actually runs the game. The `PlayGame` class will just be a wrapper for your `main` method, while the bulk of your code will go in the `CoinStrip` class.

Before you write any code, you should think about the design of your program. For programs of any significant complexity, tackling design before beginning to implement is crucial. Start with a preliminary design of the `CoinStrip` class. The design has two main parts: 1) the data structure used to represent the game state, and 2) a set of operations on the game state that are needed to play the game. A game state indicates where all the coins are, but there are multiple ways to represent this information (discussed further in the implementation section below). However you choose to represent the game state will correspond to the set of instance variables defined in the `CoinStrip` class. Operations might include those that answer questions (e.g., is a move legal? Is the game over? How many coins are there?) and those that perform actions (e.g., move a coin, print the board) – note that these examples are just possibilities, and not necessarily sufficient. Each operation you choose to support will correspond to a method of the `CoinStrip` class.

A good way to design the core data structure is to consider how the operations you will need would be handled with that design. Depending on the design of the data structure, certain operations may be easier or harder to implement. In designing the operations, consider the parameters they will take and their return values (if applicable). You probably won't be able to see all of the implications of your design choices until you've done some implementation work (i.e., written some code). Similarly, you may find once you start implementing that a previous design decision should be revised; don't be afraid to change or even completely abandon a previous design if you find that the implementation is becoming awkward or excessively complex. Design and implementation are a back-and-forth process rather than a straight-line sequence.

Your `PlayGame` class will simply be a container for your `main` method, and should not need any other methods. The `main` method should create a `CoinStrip` object representing the board and then begin prompting the players to make their moves, following the interface described in the previous section. In other words, the public methods of your `CoinStrip` class will be called from within the `main` method of `PlayGame`, which is where the game simulation is controlled. The core game simulation in `PlayGame` should essentially be a loop in which one loop iteration corresponds to one move of the game (possibly including illegal moves, depending on your design). Note that all output produced by the program (i.e., all calls to `System.out.print` or `System.out.println`) should be within `PlayGame` rather than `CoinStrip`. While methods of `CoinStrip` might produce and return `String` objects, the actual printing of such strings should only happen in `PlayGame`.

## 4 Implementation Tips

You have significant freedom in deciding how to implement your program (provided that it follows the game rules and program specification given above), but this section contains suggestions for how to approach your implementation. First, read the **Coding Design & Style Guide**, which is available at the URL below. This guide provides a summary of important aspects of design and style that you should adhere to throughout this course and beyond. Pay particular attention to the advice on modularity, since these types of decisions are more difficult to correct later on.

`https://web.bowdoin.edu/~sbarker/teaching/courses/ds/21fall/coding.php`

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. In order to test your methods, feel free to write snippets of test code in `PlayGame`, even if you will ultimately delete this test code once the program is working.

Some more specific pieces of implementation advice are given 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 (if you are familiar with these from previous Java experience), but a basic array should be fine for now.

- The most important question to ask is what the elements of the game board array actually represent. Your first inclination may be to have each array element represent a square of the board (similar to how the board is drawn visually). However, consider the following: in order to check a particular move (e.g., move coin $N$ to the left $K$ squares), you need to know both the location of coin $N$ as well as the location of the coin to its left (i.e., coin $N-1$). Using the board representation just mentioned makes performing these checks nontrivial. Think carefully about whether there is a better way to represent the board (i.e., rather than array elements representing squares of the board) that would make these checks easier to implement.

- Normally, the design of your methods (i.e., the interface of the `CoinStrip` class, as used by the `PlayGame` class) would be fully left up to you. However, since this is your first program, two specific methods are suggested: one that checks if a given move is valid (i.e., whether it is a legal next move) and another that actually performs a given move. You will still need to decide what parameters these methods will need (e.g., how a move is specified when you call the method) as well as what, if anything, they will return. Note that you will probably want to add other methods in addition to these suggested two.

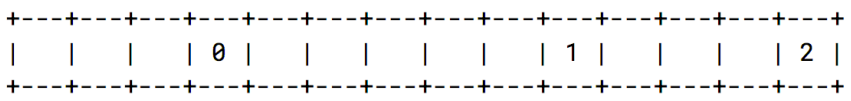
- Proper use of the `static` keyword is often a source of confusion for beginners. For this lab, we'll keep things easy for you – the **only** places you should use `static` in this lab are (1) your `main` method declaration, and (2) the declarations of any constants you use. All other methods (besides `main`) should not be marked `static`, and none of your instance variables should be marked `static`.

- 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 generated by calling the `nextInt` method on the generator object.

- 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 the `toString` method of the `CoinStrip` class.

- If you're looking for some extra practice or a bit of a challenge, you can try changing `toString` to return a fancier representation of the board, such as the one below:




  While you are not required to produce a nicer representation like this one, it's a good programming exercise to do so!

- A variant of the Silver Dollar Game is called Welter's Game, in which coins are allowed to pass each other. While this is only a slight rule change (and not one you are required to implement), think about the effect it would have on your program. You might be surprised at how significant this change would be from an implementation standpoint!

# 5 Evaluation

Your completed program (and all future programs in this course) will be evaluated along three general criteria:

1. **Correctness**: Does the program function as intended? For example, does the interface fully match the program specification, and does gameplay correctly follow the rules of the game? If you have any questions about what the "correct" behavior should be, ask your instructor (but chances are that the answer is already in this writeup!).

2. **Design**: Is the program well-designed? Are the choices of data structures sound? Is common functionality extracted into methods to avoid duplication? Are methods specific and appropriately modular? While this is far from an exhaustive list, these kinds of design questions are of critical importance, even in the context of a fully correct program.

3. **Style**: Is the code properly and consistently indented? Are variables descriptively and appropriately named? Are constants used when appropriate? Are classes and methods appropriately documented? These kinds of questions do not affect program correctness but greatly affect how readable and editable the program is.

In self-screening your program before submission, your primary reference for correctness-related issues should be your lab writeup (i.e., this document). For design and style issues, your primary references should be the Coding Design & Style Guide (read it again!), your own good sense, and any specific guidance provided by your instructor (e.g., feedback provided on past labs).

**Lastly, make sure that your name is included in the `CoinStrip` and `PlayGame` classes, and make sure that your project directory is properly named as described in the next section. Points may be deducted if these guidelines are not followed!**

# 6 Submitting Your Program

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

1. Save your program and quit your IDE (e.g., BlueJ or Eclipse).
2. **Rename your project folder (which is the folder containing your `.java` files and possibly a few other files as well) so that it is named `username-lab1`, where `username` is replaced by your Bowdoin username. For example, if your username is `jdoe`, you should rename your folder `jdoe-lab1`. Do not include anything else in the folder name!**
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 `jdoe-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 `Browse Local Files` to locate and attach your `.zip` archive. Don't write any comments in the comment section, as your instructor will not see them. Once you've attached your `.zip` archive, click on `Submit` to complete your submission.

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 (and not available if you log into any other machine on campus). Two good options to keep your lab files available on any machine are Dropbox (or any similar service) or in your folder on the Bowdoin `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`.