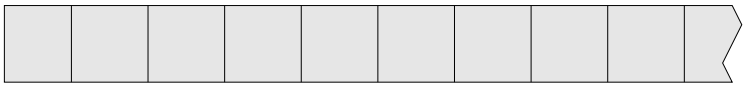
Computer Science CS136 (Fall 2021)
Duane Bailey & Aaron Williams
Laboratory 1
The Silver Dollar Game

Objective. To implement a simple game using `Vectors` or arrays.

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



The "coin strip" used in the Silver Dollar game.

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.

A possible starting position for the Silver Dollar game.

The goal is to move all the n coins to the leftmost n squares of the paper. This is accomplished by players alternately moving a single coin, constrained by the following rules:

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

If multiple people are playing, then the last person to move is the winner.

This week we will be thinking about how to represent the "coin strip" at the center of this challenge. There are many choices of how we might represent the state of the coin strip; your job is to identify one that seems appealing and make it the basis of your implementation. When your implementation is finished, you can test it using two existing applications—a solitare "puzzle", and a two-person "game".

Preparation. As we begin thinking about implementing various data structures this semester it may be useful to leverage the data structures found in the `structure` package. To install this package, you should clone the *Java Structures* resources into a subdirectory of your `cs136` directory:

```
cd ~/cs136
git clone ssh://22xyz3@lohani.cs.williams.edu/~bailey/js.git
```

where 22xyz3 is replaced with your CS user name. You will be asked for your password. When the clone is finished, there is a new folder, `js`, that contains the `Java Structures` text, in `.pdf` form, as well as a Java library or "jar file" (*java archive*) that contains all the classes discussed in the text.

We must tell Java where this library is located. This is accomplished by updating the shell environment variable `CLASSPATH`. This variable contains a colon-separated list of directories and `jar` files where Java classes may be found. The instructions found in the file `INSTALL.txt` in the `js` directory will help you add this line to your shell startup script:

```
export CLASSPATH=.:~/cs136/js/bailey.jar:$CLASSPATH
```

For instructions on how to do this, please read the file `INSTALL.txt`. Once the `structure` package is installed on a machine, it need not be installed again. (Remember that your files are shared between the Unix machines, but not the Mac machines.)

This Week's Tasks.

This week we encourge you to read what must be done *before* you actually start designing your implementation of the `CoinStrip` class. We expect the public interface to your class to be thoughfully implemented and designed in a way that reduces the complexity of your methods. Generally: How do you represent the *state* of the `CoinStrip`, and how *hard* or *complex* is it to support the required methods given your representation?

Here is what we expect you to do:

1. Clone your lab1 repository:

   ```
   cd ~/cs136
   git clone https://evolene.cs.williams.edu/cs136-labs/22xyz3/lab1.git
   ```

   replacing 22xyz3 with your CS username. This will create a subdirectory, `lab1` in your `cs136` directory.

2. Change to the `lab1` directory and look around:

   ```
   cd ~/cs136/lab1
   ls
   ```

   Notice that we have given you two different applications that will help you test your `CoinStrip` implementation:

   - `Puzzle.java` plays a solitare game that will allow the user to experiment with different approaches to moving the coins.
   - `Game.java` allows two different players to alternate turns moving the coins on a single `CoinStrip`.

   When these applications start, the computer has set up a random coin strip with at least 3 coins. Before each move, the state of the coin strip is printed. Then the player (or players) are allowed to enter a move. A move is specified by a coin number (a value between 0 and $n-1$ for a n coin challenge) and the number of squares to move the coin to the left. If the move is illegal, the player is repeatedly prompted to enter a revised move. Once a legal move has been entered, the state of the coin strip is updated. Between turns the computer checks the board state to determine if the challenge is finished.

3. We've outlined a minimal class, `CoinStrip`, for you to implement, and possibly extend. At a minimum, the following methods should be coded, based on your choice of representation:

   - `public CoinStrip()`. This is the constructor for the `CoinStrip` object. At a minimum it should place three coins on a strip in a non-winning position. You will want these randomly placed, within reason. A good way to generate random numbers is to use the `Random` class from `java.util`. At the top of your program:

```
import java.util.Random;
```

Within a class, to construct a random number generator, g, use:

```
g = new Random();
```

This generator can then be asked to generate random integers with the `nextInt(n)` method, which returns one of the `n` values from 0 to `n-1`. To roll a number between 1 and 6 (as with a die), you could use:

```
value = 1+g.nextInt(6);
```

Ideally, you make use of a single generator to support the randomness in a single object. In other words, to generate many random numbers for your `CoinStrip`, create one generator, and call its `nextInt()` method many times.

- `public int numCoins()` and `public int numSquares()`. These accessor methods return the number of coins in the coin strip, and the length of the strip, in squares, necessary to hold the coins. As the game progresses, the number of coins does not change, but the number of squares decreases as the rightmost coin moves left.
- `public int square(int i)`. Given a location of a square (let's agree that the leftmost square is zero, and they increase to the right), this method describes the content of the square: it returns -1 if it is blank, or the number of the coin, if it is occupied. This method should work for any non-negative value of `i`.
- `public boolean gameOver()`. This routine determines if the n coins occupy the leftmost n squares. If so, it returns `true`, otherwise it returns `false`.
- `private boolean legalMove(int coin, int delta)`. This verifies that the indicated move is valid. Is the coin number a valid coin? Does the coin have at least `delta` spaces to its left? If the move is valid, it returns `true`, otherwise `false`. We expect you will find this useful in implementing the `move` method.
- `public boolean move(int coin, int delta)`. This attempts to move the indicated coin `delta` spaces to the left. If the move is legal, the state of the coin strip is updated and it returns `true`. Otherwise, it returns `false`.

We have given you an implementation of `toString()`, the method that generates a `String` representation of the `CoinStrip`, but notice that it depends on several of the above methods being implemented correctly.

4. As you evaluate your design, you might consider the following:

   (a) Are you going to use a Java array, or will you use a `Vector` from the `structure` package? What is your motivation behind this design decision?

   (b) Does your representation store only the *sufficient* information to represent the coin strip? Does your representation store *more* information than is necessary?

   (c) How hard are the accessor methods to write, given your design?

   (d) Is it easy to test for a legal move?

   (e) Is it easy to test if the game is done?

(f) Is it easy to generate coin strips that are not immediate wins?

As you develop an approach, you might write up your notes as an informal "design document." You might draw pictures to help you understand how each of your private instance variables supports the structure. This document will likely prove useful as you progress through the implementation.

5. We've given you a set of methods that are clearly important, but you may think of others, as well. Feel free to add those, but make sure they don't detract from the basic functionality of the class.

6. Make sure that your `CoinStrip` class works with the main methods of `Puzzle` and `Game`. If you wish, you can modify these applications to demonstrate any extensions you implement.

Extras. Here are some ways that you can extend the functionality of these classes:

1. It may be interesting to have the number of coins in a `CoinStrip` vary. For example, you could have the `CoinStrip` include 3 coins with probability 50%, 4 coins with probability 25%, and so forth.

2. Since this game is very similar to the game of *Nim*, where the size of the move is limited, it might be useful to specify the range of sizes of gaps between adjacent coins.

3. The computer could occasionally provide helpful hints to the players. What opportunities appear easy to recognize? How would you deliver hints between the `CoinStrip` and the applications that use it?

4. The applications are pretty simple, as they stand. Modify the application so that there is one additional constraint on the user's move: the distance must be three or less. With care on constructing `CoinStrips`, this will increase the complexity of the puzzle and game.

Submitting Your Work. To get credit for this week's lab make sure that you've added, committed, and pushed the files `CoinStrip.java`, `honorcode.txt` (containing your signature), and, if necessary, `Puzzle.java` and `Game.java`. Remember: if you type

```
git push
```

the response `"Everything up to date"` is an indication that your work has been turned in on `evolene`.

⋆