Assignment 2: Freecell (Part 1)

Due Friday by 8:59pm **Points** 0

The Model

Starter files: **code**

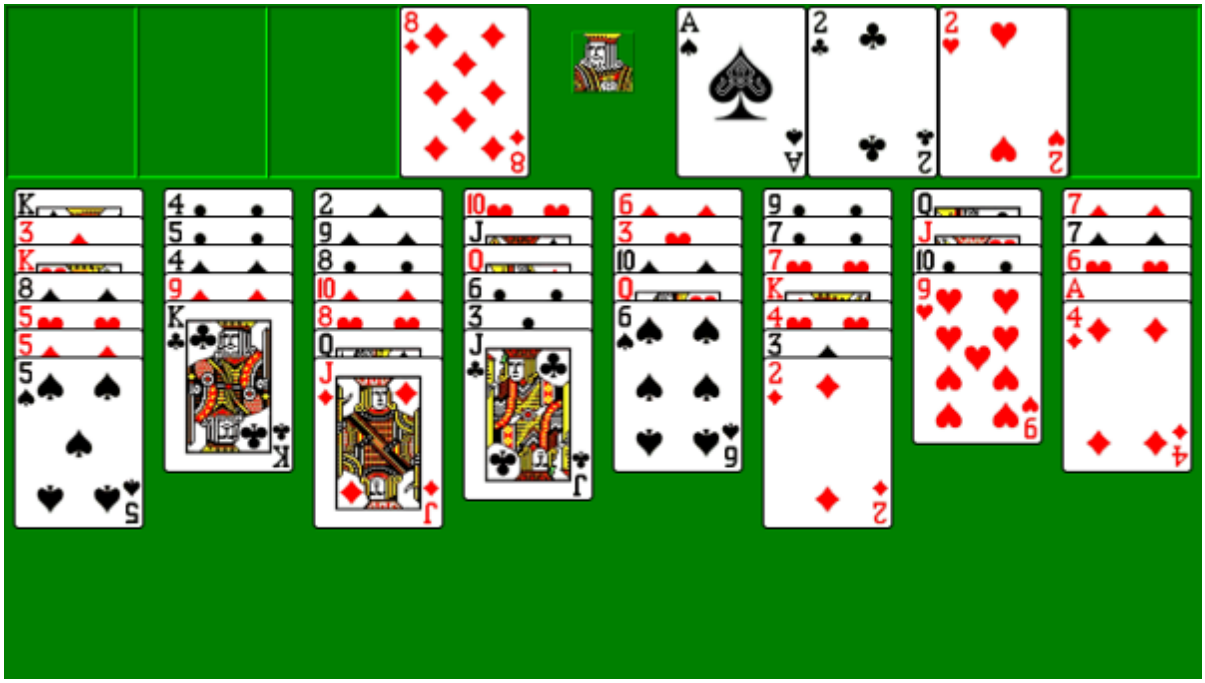
Note: The description may make assignments seem longer than they are. Distilling the description to make a list of all the things you are actually supposed to do will go a long way in having a good plan to tackle it. Read the description several times to confirm this list before acting on it!

1 Purpose

The primary goal of this assignment is to practice implementing an interface based on given specifications, and by choosing an appropriate data representation that helps in providing the functionality promised by the interface.

2 The game of Freecell

2.1 Context



(https://play.google.com/store/apps/details?id=net.runserver.freecell_free)

In the next three homeworks, you will implement the popular single-player game "Freecell." Freecell is a Solitaire-type game, which uses the regular deck of 52 suit-value cards. There are four suits: clubs (♣), diamonds (♦), hearts (♥), and spades (♠). Hearts and diamonds are colored red; clubs and spades are

colored black. There are thirteen values: ace (written A), two through ten (written 2 through 10), jack (J), queen (Q) and king (K). In Freecell, aces are considered "low," or less than a two. (In other games, they're considered "high," or greater than a king.)

The game play is divided among three types of *card piles* . First, there are four *foundation* piles, one for each suit (shown in the top right in figure above), and indexed 1 through 4. These four piles are initially empty, and the goal of Freecell is to collect all cards of all suits in their respective foundation piles. A card can be added to a foundation pile if and only if its suit matches that of the pile, and its value is one more than that of the card currently on top of the pile (i.e., the next card in foundation pile 2 in the figure above can only be 3♣). If a foundation pile is currently empty, any ace can be added to it: there is no required ordering of suits in the foundation piles. Note that in the figure above, the 2♣ has an A♣ underneath it, and the 2♥ has an A♥ underneath it, though these aces are not visible.

The second type of pile is the *cascade pile* (the eight piles in the bottom of figure above), also indexed starting from 1. Usually a game of Freecell has eight cascade piles, but our game will allow as few as four. Because the initial deal of the game is shuffled (see below), a cascade pile may initially contain cards in any order. However, a card from some pile can be *moved* to the end of a cascade pile if and only if its color is different from that of the currently last card, and its value is exactly one less than that of the currently last card (e.g. in the figure above, the next card in cascade pile 1 can be 4♦ or 4♥ while the next card in cascade pile 3 can be 10♠ or 10♣). This sequence of decreasing-value, alternating-color cards is called a *build* . (Different variants of Freecell, or other solitaire games, differ primarily in what builds are permitted.)

Finally, the third type of pile is the *open pile* (top left in figure above). Usually a game of Freecell has four open piles, but our game will allow as few as one. An open pile may contain at most one card. An open pile is usually used as a temporary buffer during the game to hold cards.

2.1.1 Game play

Play starts by dealing the full deck of 52 cards among the cascade piles, in a round-robin fashion (e.g. in a game with eight cascade piles, pile 1 will have card indices 0, 8, 16,..., pile 2 will have card indices 1, 9, 17, ... and so on (assuming card indices in the deck begin at 0). The player then moves cards between piles, obeying the rules above. The objective of the game is to collect all cards in the four foundation piles, at which stage the game is over.

In this homework you will write the model for this game. The model will maintain the game state and allow a client to start a game and move cards.

3 Building Freecell

3.1 Cards

Start by modeling a card in the game of Freecell. You are free to name the class and its methods whatever you want.

3.2 The Freecell operations

In order to play the game, one would expect the following operations: get a valid deck of cards, start a new game, make a move and return several aspects of the current state of the game. The operations related to the state of the game are specified and documented in the provided `FreecellModelState<K>` interface. The provided `FreecellModel<K>` interface extends this interface to provide a complete set of operations for the model of this game. In both these interfaces, `K` is the card type (because we do not know in advance what you may choose to name your card type). You are **not** allowed to change the interface in any way!

A short explanation of some of the methods in the interface is given below. For a complete description read the documentation provided in the two interfaces (generate Javadoc for better readability).

- `getDeck` should return a valid deck of cards that can be used to play a game of Freecell.
- `startGame(List<K> deck, int numCascades, int numOpens, boolean shuffle)` takes a deck and starts a new game of Freecell with the specified number of cascade and open piles. If shuffled is `true` it shuffles the specified deck before dealing, otherwise it deals the deck as-is. This method throws an `IllegalArgumentException` if any of the parameters are invalid.
- `move(PileType sourceType, int sourcePileNumber, int cardIndex, PileType destinationType, int destPileNumber)` is called to move a card according to the rules of the game. Specifically, it moves cards beginning at index `cardIndex` from the pile number `sourcePileNumber` of type `sourceType` to the pile number `destPileNumber` of type `destinationType`. `PileType` (provided to you) is an enumerated type that is helpful in specifying the type of pile.

All indices and pile numbers in this method start at 0 --- e.g. in the figure above, to move 4♦ in cascade pile 8 to cascade pile 1 this method would be called as `move(PileType.CASCADE, 7, 4, PileType.CASCADE, 0)`. This method throws an `IllegalStateException` if the game has not started, and an `IllegalArgumentException` if the game has started but the move is not possible.

- `isGameOver` returns `true` if the game is over, and `false` otherwise. The game is over only when all foundation piles are full, i.e., the game has been won.

3.3 Examples

You must implement the `FreecellModel` interface in a `SimpleFreecellModel` class. Please review the testing recommendations. Since you are testing the public-facing behavior of this interface, following those guidelines means that you should ***not*** place this testing code in the `cs3500.freecell.model.hw02` package, but rather place it in the *default* package. Nothing in your tests should rely on implementation details at all.

3.4 Your Implementation

Implement the `FreecellModel<K>` interface in a class called `SimpleFreecellModel`. Unlike the provided interface, this implementation should be specific to your card implementation (i.e. it should not be

genericized). It should implement all operations according to their specifications. This class should be placed in the `cs3500.freecell.model.hw02` package. You will need to:

> *To do:*
>
> - Design a representation for the game skeleton. Think carefully about what fields and types you need to represent the game state, and how possible values of the fields correspond to game states.
> - **Cards:**
>   - A correct deck of cards for Freecell contains 52 cards with ace, 2, ..., jack, queen, king in each of clubs, diamonds, hearts and spades) as a `List` of card objects. Thus a deck is **invalid** for this game if it does not have 52 cards, or if there are duplicate cards, or if there are invalid cards (invalid suit or invalid number).
>   - A client for your Card implementation should be able to use its `toString` method to print them. Cards should be printed as their value, followed by the suit symbol above: for example, "A♦" or "3♣". You may want to copy and paste the suit symbols from this Web page into IntelliJ.
> - **Starting the game:** The `startGame` method should start a new game of Freecell by distributing the deck in round-robin fashion as described above (in Section "Game Play"). It is possible that some cascade piles have more cards than others, and this is OK. This method should first check if the inputs are valid (valid deck, at least 4 cascade piles and at least 1 open pile). If shuffle is `true` then the cascade piles should look different compared to if it was `false`.
> - **Move:** The `move` method should work to move only one card at a time. This means that only the last card in a cascade pile can be moved to another pile. Any other kind of move is invalid for this implementation.
> - **Constructing the model:** Your class should have a `public` default constructor (i.e., with zero arguments) for your class, and also any other constructors you think are needed. Reminder: A constructor is a good place to initialize any instance variables.
>
> Be sure to properly document your code with Javadoc as appropriate. Method implementations that inherit Javadoc need not provide their own unless they implement something different or in addition to what is specified in the inherited documentation.

3.5 View

Our game should have some way of showing us the game board during game play. You have been provided with a `FreecellView` interface that represents a view. In this assignment, you should implement a class called `FreecellTextView` in the `cs3500.freecell.view` package. This class should contain the following:

To do:

- It should have a constructor with exactly one argument of `FreecellModelState<?>` type. This constructor can be called with the model object that provides this view all the methods it needs to query the model and print the board.

Note that the generic type is `<?>`, and not `<K>`. Unlike the model, the view need not be dependent on a card type, because the view is only concerned with printing cards and not operating on them. The <?>, called a wildcard, achieves this. It means that this view can work with *any* implementation of the `FreecellModelState` interface, not just your specific `SimpleFreecellModel` dependent on your specific card type. **Read about wildcards here. (https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html)**

- The `toString` method of this class returns a `String` which may be used to print the board, in the format illustrated below. The details of the output are described in the Javadoc for this method in the provided `FreecellView` interface. The following shows the output corresponding to the screenshot above:

```
F1: A♠
F2: A♣, 2♣
F3: A♥, 2♥
F4:
O1:
O2:
O3:
O4: 8♦
C1: K♠, 3♦, K♥, 8♠, 5♥, 5♦, 5♠
C2: 4♣, 5♣, 4♠, 9♦, K♣
C3: 2♠, 9♠, 8♣, 10♦, 8♥, Q♠, J♦
C4: 10♥, J♠, Q♦, 6♣, 3♣, J♣
C5: 6♦, 3♥, 10♠, Q♥, 6♠
C6: 9♣, 7♣, 7♥, K♦, 4♥, 3♠, 2♦
C7: Q♣, J♥, 10♣, 9♥
C8: 7♦, 7♠, 6♥, A♦, 4♦
```

NOTE: The string you return should not have a newline at the end of the last line, and there should be no spaces after the colons for O1, O2 or O3. (Try selecting and highlighting this text in your browser, to see exactly where the lines end.)

If a game has not begun, `toString` should return an empty string.

File Hw02TypeChecks.java contains a small class designed to help you detect when your code may not compile against the grading tests. In particular, if your project cannot compile while Hw02TypeChecks.java---unmodified!---is a part of it, then it won't compile for the course staff either.

3.6 Testing

You are expected to test all the code that you write. Please place all your tests in the default package. Please read the **posted testing recommendations**

(https://northeastern.instructure.com/courses/79060/pages/How%20To%20Think%20About%20Testing) again.

4 List of Deliverables

- Your card class(es)
- The interfaces (`FreecellModelState.java` , `FreecellModelState.java` and `FreecellView.java`)
- The `SimpleFreecell` implementation of the `FreecellModel` interface
- The `FreecellTextView` implementation of the `FreecellView` interface
- any additional classes you saw fit to write
- tests in one or more suitably named JUnit test classes

5 Grading Standards

For this assignment, you will be graded on

- whether your code implements the specification (functional correctness),
- the appropriateness of your chosen representations,
- the clarity of your code,
- the comprehensiveness of your test coverage
- how well you have documented your code
- how well you follow the **Design Principles** (https://northeastern.instructure.com/courses/79060/pages/Design%20Principles%20Master%20List) where applicable
- how well you follow the **style guide** (https://google.github.io/styleguide/javaguide.html) .

6 Submission

Wait! Please read the assignment again and verify that you have not forgotten anything!

Please compress the src/ and test/ folders into a zip file and submit it. After submission, check your submitted code to ensure that you see two top-level folders: src/ and test/. If you see anything else, you did not create the zip file correctly!

Please submit your assignment to https://handins.ccs.neu.edu/ by the above deadline. Then be sure to complete your self evaluation by the second deadline.