Assignment 2: Playing with Cards, Part 1: The Model

Due dates:

- **Examplar submissions: Saturday, Sept 23 at 8:59pm**
- **Implementation: Thursday, Sept 28 at 8:59pm**
- **Self-evaluation: Friday, Sept 29 at 11:59pm**

Starter files: `code.zip`

Note: The description may make assignments seem longer than they are. Distilling the description to make a lis[illegible]osed to do will go a long way in having a good plan to tackle it. Rea[illegible]rm this list before acting on it!

1 [illegible]

The[illegible]ractice implementing an interface based on given specifications, and[illegible]sentation that helps in providing the functionality promised by the[illegible]

**Plea[illegible]n the `cs3500.klondike.model.hw02` and `cs3`[illegible]e that the model package refers to `hw02`, and the view does not. (See[illegible]

There will be three submissions for this assignment:

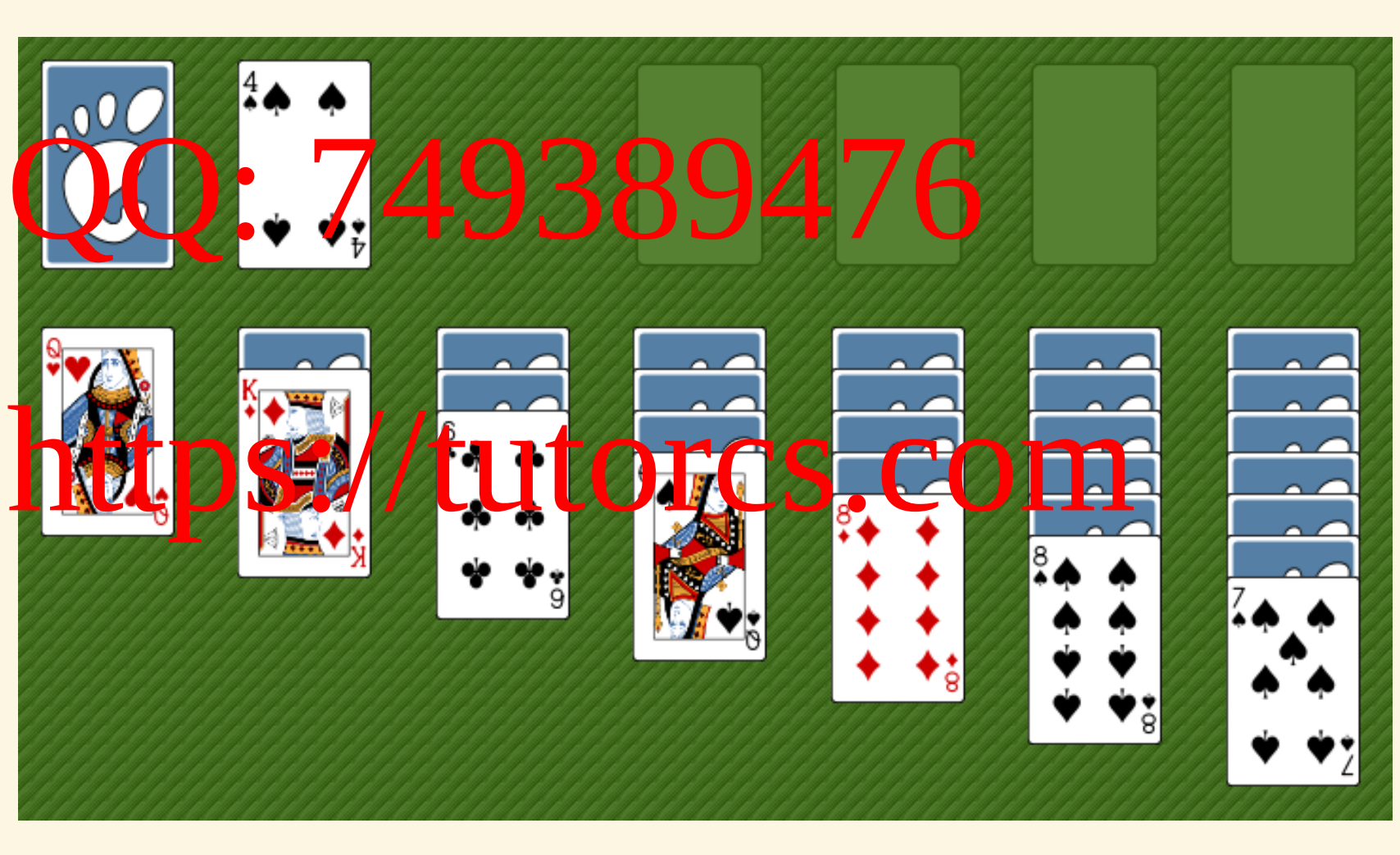
- An *Examplar* submission, early in the week, where you will submit a small set of examples designed to probe *our* implementations of the game and find several simple possible bugs or points of confusion. See Examplar for more information about the goals of Examplar assignments.
- Your *actual implementation* and full test suite
- A *self-evaluation*, due *one day plus three hours later* than the actual implementation, where we will ask you to reflect on your implementation and testing choices.

A reminder about late days: Each submission is a *distinct* submission, and each one will independently use up your late days if you submit after the deadline. (Submitting late for Examplar does not mean you automatically get a "free" late day for the assignment as well: submitting both parts late will use *two* late days.) The 27-hour window for the self-evaluation is deliberate, so that even if you submit your assignment late, you can still submit your self-evaluation on time.

Read below for more details of each submission.

2 The game of Klondike

2.1 Context



In the next three assignments, you will implement the common single-player card game called "Klondike." The image above shows a Klondike layout at the start of a game.

2.1.1 Game Play

Klondike uses the standard suit-value playing cards for its game. There are four suits: clubs (♣), diamonds (♢), hearts (♡), and spades (♠). Hearts and diamonds are colored red; clubs and spades are colored black.[1] There are thirteen values: ace (written `A`), two through ten (written `2` through `10`), jack (`J`), queen (`Q`) and king (`K`).

There are three areas of the game play. First, there are the *foundation* piles (in the upper right region of the picture above). These piles are initially empty; the goal of the game is to move all the cards into the foundation piles, such that each foundation pile contains a single suit of all thirteen cards, in order, starting from an ace and ending at a king. The second group of piles are the *cascade* piles (in the lower half of the picture above). Play proceeds by moving cards among piles, following the rules below, in order to reveal all the cards and move them to the foundation piles. Finally, there is the *draw pile* (in the upper left of the picture), which contains all the other cards that are not yet in the cascade or foundation piles: the top few cards of this pile are revealed.

In a standard game, there are 52 cards: one complete set of all possible values in each possible suit. Our game will be more flexible: our deck should consist of equal-length single-suit runs of consecutive values starting from Ace, and there can be as many runs in the deck as desired. (For example: it is valid to have three complete sets of hearts and spades; or to have one set of each suit from Ace through Five. But it would be invalid to have all thirteen clubs and only the Ace through Five of diamonds.) There should be as many foundation piles as there are aces in the deck. A standard game of Klondike deals the cards into seven cascade piles, but our game will allow any (positive) number of piles, provided there are enough cards in the deck to deal them out completely. (For example, a standard deck can be dealt into at most nine cascade piles; two decks can be dealt into at most thirteen piles; etc.) The draw pile may show one or more cards, however, the player may only use the first card (i.e. the cards must be used in order).

2.2 Game play

Play starts by dealing the cards into the cascade piles, from left to right and top to bottom. All the cards are face down, except the bottom-most card of each pile. All remaining cards are placed face down into the draw pile, whose top few cards are turned over.

The face-up cards in the cascade piles must form *builds*: they must be in consecutive, descending order from top to bottom, and must alternate in color.

A player may make one of several moves:

- Move the bottommost card of a cascade pile to a foundation pile. If the foundation pile is empty, the card must be an Ace. Otherwise, the card must be the same suit as the existing top card of the foundation pile, and be exactly one higher in value than the existing card.
- Move any number of face-up cards from one cascade pile to another. If the destination pile is empty, the topmost card of the build must be a King. Otherwise, the topmost card of the build must be the opposite color and one lower in value than the bottommost card of the destination pile — in other words, after the move, the destination pile should still have a legal build on it. If the move leaves the source pile with no face-up cards, then turn over the bottommost card of the source pile (if there are any left).
- Move the first draw card to a foundation pile. This must follow the same rules as moving a cascade-pile card to a foundation pile. After this move, reveal the next draw card (if any).
- Move the first draw card to a cascade pile. This must follow the same rules as moving a single card from one cascade pile to another. After this move, reveal the next draw card (if any).
- Discard the first draw card to the bottom of the draw pile, and reveal the next draw card. (In our version of this game, the draw pile can be reused indefinitely.)

The *score* of the game is the number of cards moved into the foundation piles.

The game is over when there are no legal moves remaining.

3 Building Klondike

In this assignment you will design the model for this game. The model will maintain the state of the game and update itself when a client specifies moves. You are *not* required to make the game playable by a user at this point: only you-the-programmer can manipulate the model right now, and there is no mechanism yet for you-the-player to actually specify moves and play the game.

3.1 Cards

Start by modeling a card in the game of Klondike. You are free to name the class and its methods whatever you want, but it must implement the `Card` interface given to you. Your card implementation should behave like a proper "good citizen of Java", and implement its own `toString`, `equals` and `hashCode` methods. (See below for some hints.) The `toString` method should render the cards as described above: e.g. `"A♡"` or `"10♠"`, etc.

3.2 Expected operations

In order to play the game, the client would expect the following operations: start a new game, make a move, get the current state of the game, get the current score and know when the game has ended. These operations have been specified and documented in the provided `KlondikeModel` interface. You are **not** allowed to change the interface in any way!

A short explanation of most of the interface follows (the explanation here supplements the documentation provided in the interface):

- The `KlondikeModel` interface itself uses the `Card` interface to describe cards — you will need to implement that `Card` interface to represent cards, as described above.
- `getDeck()` should return a deck containing all the cards that your model knows how to work with.
- `startGame(List<Card> deck, boolean shuffle, int numPiles, int numDraw)` follows the description above, and lets the caller specify the number of piles of the cascade and the maximum number of cards to be displayed in the draw pile at any given time. Additionally, to make the game more easily testable, this method supplies a deck of cards to be used, and specifies whether the model should shuffle the cards before dealing them, or should use the order given by that deck.
- `movePile`, `moveDraw`, `moveToFoundation`, `moveDrawToFoundation` and `discardDraw` implement the five player moves described above. Whenever present, any indices in the parameters are assumed to be zero-based, counting from the left.
- At any point during the game, the cascade piles "fit" into a rectangle that is `getNumPiles()` piles wide and `getNumRows()` rows tall.
- `getPileHeight(int pileNum)` returns the current height of the given cascade pile. (Naturally, it must always be a value between 0 and `getNumRows()`.)
- `getCardAt(int pileNum, int card)` returns the card at the given coordinates, if it is visible. To check if it is visible, use `isCardVisible` with the same arguments.
- `getCardAt(int foundationPile)` returns the top card of the given foundation pile. (Naturally, the index must be between 0 and `getNumFoundations()`.)
- `getNumDraw()` returns the *maximum* number of draw cards visible at any time. The *actual* draw cards visible at any time can be retrieved with `getDrawCards()`.
- `isGameOver()` returns `true` if the game is over, and `false` otherwise.
- `getScore()` returns the current score in the game.

3.3 Examplar

The starter code above provides you with the `KlondikeModel` and `Card` interfaces and a stub implementation of the `BasicKlondike` class. *You should not change these files at all.* It also provides you with an empty `ExamplarModelTests` class for you to fill in with your example test methods.

Just as functions and methods deserve purpose statements in your implementation, so too your test methods deserve explanation. Either name your test methods descriptively (e.g. `testMovePileCardToEmptyFoundationWorksCorrectly`), or leave a comment above the method that briefly explains what the scenario is that you're probing and what bug you might be trying to detect (e.g. "The rules specify that only Aces can be moved to empty foundations, so this test should throw an exception.") These comments will be helpful both to your graders when they're trying to understand what you've tried so far, and to you as a reminder of scenarios that your model should handle properly.

Hints:

- You can trust that the output of all observation methods returns meaningful and accurate information about the current state of the game.
- `startGame` is guaranteed to work as intended.
- Any of the mutator methods are potentially buggy; you should focus your attention on them. Likewise, you should ensure there are no unintended mutators.
- Take advantage of the flexibility in the model interface to create small, "rigged" games. Use the mutators to "drive" the model towards a scenario you want to test, and then check that it's produced the expected scenario.

3.4 Your Model Implementation

Implement the `KlondikeModel` interface by filling in the stubs in the `BasicKlondike` class:

1. Design a suitable representation of this game. Think carefully about what fields and types you will need, and how possible values of the fields correspond to game states.

2. Instantiating the game: Your class should define at least one constructor with zero arguments, which initializes your game into a state that's ready for someone to call `startGame` and begin playing. You may define whatever other constructors you wish; consider carefully all the methods you are expected to implement, and design your code to avoid as much duplication as possible. Keep in mind that a client should not be able to start a game without calling the `startGame` method!

3. Encapsulation: Your `BasicKlondike` class should not have *any* public fields, nor any public methods other than constructors and the public methods required by the `KlondikeModel` interface.

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 Viewing the model

Our game should have some way of showing us the game board during game play. You have been provided with an empty `KlondikeView` interface that represents a view — we will add meaningful methods to this interface later. In this assignment, you should implement a class called `KlondikeTextualView` in the `cs3500.klondike.view` package.

```
public interface TextualView {

}

public class KlondikeTextualView {
  private final KlondikeModel model;
  // ... any other fields you need

  public KlondikeTextualView(KlondikeModel model) {
    this.model = model;
  }

  // your implementation goes here
}
```

1. Your class should at least have a constructor with exactly one argument of type `KlondikeModel` — this model provides all the information the view needs in order to be rendered.
2. The `toString()` method of this class returns a `String` that may be used to display the board. Here is an example rendering of a recently-started game; your `toString()` method should reproduce this:

```
Draw: 8♣, 6♢, K♡
Foundation: <none>, A♡, 3♢, <none>
 A♣  ?  X  ?  ?  ?  ?
    2♠     ?  ?  ?  ?
           ?  ?  ?  ?
          5♡  ?  ?  ?
             6♡  ?  ?
                7♢  ?
                   7♠
```

The first line shows the draw cards, with the "first" card (the one that can be moved or discarded) being on the left.
The second line shows the foundation piles. Each empty pile is shown as `<none>`, and non-empty foundations show the topmost (highest-value) card.
Beneath that are the cascade piles. Each pile is three characters wide, and cards are right-aligned within that column. (Each row of the display should therefore be exactly `3 * getNumPiles()` characters wide.) Face-down cards are shown as `?`. Empty piles are shown as a single `X` (here, the third pile is empty). Every line should end with a newline character, ***except*** the final line — in this example, the first character of output is the `'D'` of `"Draw"` and the final character is the `'♠'` of the `"7♠"`.

3.6 Testing

After you've submitted your Examplar examples, you may well need to add *more* tests, that might assess whether your model implementation passes *additional* checks that you didn't think of initially, and that possibly are not entirely specified by the interface (e.g., that `getDeck` returns cards in a particular order).

To do that, you should create *two new test classes*. One of them should go alongside the `cs3500.klondike.ExamplarModelTests` class, and it should contain any new tests you've thought of that didn't make it into your Examplar submission, but that nevertheless are testing properties of the *public* model interface. To test implementation-specific details, you should creeate one last test class that you would place in the `cs3500.klondike.model.hw02` package itself, so that you can check package-private implementation details if needed.

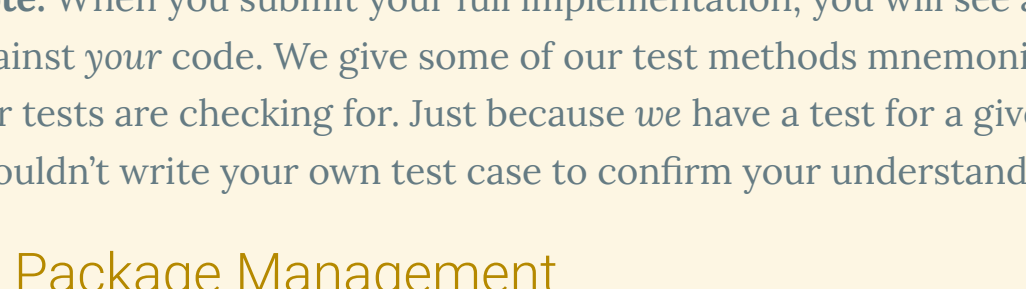
Be mindful of which test cases you place in which test class! *Technically*, you could run all the tests from a single class. But using multiple classes like this helps convey to the reader of your code some of your thought processes behind each test: the reader should understand the examples first, then look at the tests of public behavior, and finally look at implementation-specific fiddly details.

Note: When you submit your full implementation, you will see automated tests that *we* wrote and run against *your* code. We give some of our test methods mnemonic names, so that you can try to deduce what our tests are checking for. Just because *we* have a test for a given scenario, though, does not mean that you shouldn't write your own test case to confirm your understanding!

4 Package Management

To make sure that your packages are in the correct layout, you should tell IntelliJ to do the following. Do this early, before you've written much code, to ensure that your files wind up in the right locations automatically, instead of having to fix it afterward:

- When you create a new project, you should see something like this:

  File Edit View Navigate Code Analyze Refactor Buil
Demo
Project
Packages
Demo (C:\Users\Ben\IdeaProjects\Demo)
.idea
src
Demo.iml
External Libraries

  Notice that the `src` directory is marked blue, which means IntelliJ believes that this directory contains the source files of your project. *If it isn't marked blue*, you need to tell IntelliJ that it should be: right-click on the `src` folder and select *Mark Directory As -> Sources root*. To create a new package, right-click on the `src` directory, select *New -> Package*. In the dialog box that pops up, enter the new package name
- To create new files within a particular package, right-click on the package folder and select *New -> Java Class*. If you want to create a new file in the *default* package, then select the `src` directory itself.
- To create a test directory, right-click on the *project* itself, and select *New -> Directory*. In the dialog box that pops up, enter "test" as the name. Right-click on the directory, select *Mark Directory As -> Test Sources root*. Henceforth, you should add any test classes in this folder. See the tutorial video for a demo of this.
- The `src/` and `test/` directories can parallel each other in structure. However, keeping your sources and tests separated is always a good idea, so you don't inadvertently release your tests as part of your source!

5 What to submit

- For Examplar: submit a properly-structured zip containing
  - *only* your `ExamplarModelTests.java` file
- For your implementation: submit a properly-structured zip containing
  - The model interface (`KlondikeModel.java`)
  - Implementation of the model interface (`BasicKlondike.java`)
  - Implementation of the view (`KlondikeTextualView.java`)
  - Any additional classes you saw fit to write
  - All your tests (including Examplar) in one or more JUnit test classes

Again, please ensure all of your project's *sources* are in the `cs3500.klondike.model.hw02` and `cs3500.klondike.view` packages, accordingly. Please ensure that your project's *test cases* are in the packages explained above. Note that the model package refers to `hw02`, and the view does not. The autograder will give you an automatic 0 if it cannot compile your code!

6 Grading Standards

For this assignment, you will be graded on

- whether your code implements the specification (functional correctness),
- the appropriateness of your chosen representation,
- the clarity of your code,
- the comprehensiveness of your test coverage
- how well you have documented your code
- how well you follow the style guide.

7 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 do not include your `output/` or `.idea/` directories — they're not useful!**

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.

1 In the textual representation here, "red" suits are simply shown as hollow shapes, while "black" suits are filled-in.