AP Computer Science with Data Structures

Lab: Solitaire

In this lab, you will use Java's `Stack<E>` class to implement the classic Klondike Solitaire card game. Go ahead and download the files Solitaire.java and SolitaireDisplay.java to a new directory. You should also download cards.zip and extract the contents (card images) to a subfolder of that same directory.

Exercise 1: Cards

Create a class called `Card`, which will represent a single playing card with three attributes:

`rank` (an `int` from 1 to 13, where 1=Ace and 13=King)
`suit` (a `String`, where `"c"`=♣, `"d"`=♦, `"h"`=♥, and `"s"`=♠) `isFaceUp` (a `boolean`)

Write a constructor that takes in a rank and suit. All cards will be face down by default.

Implement each of the following `public` methods:

```
int getRank()
String getSuit()
boolean isRed()      //returns true if ♦ or ♥
boolean isFaceUp()
void turnUp()        //makes the card face up
void turnDown()      //makes the card face down
```

Finally, implement the method `getFileName`. On a mac if the `Card` is face down, `getFileName` should return `"cards/backapcsds.gif"`, assuming you unzipped the card images to a subdirectory named `"cards"`. (Notice that we're using the escape sequence for the backslash character.) If the `Card` is face up, then return:

`"cards/ac.gif"` for A♣
`"cards/2d.gif"` for 2♦
`"cards/9h.gif"` for 9♥
`"cards/ts.gif"` for 10♠
`"cards/jc.gif"` for J♣
`"cards/qd.gif"` for Q♦
`"cards/kh.gif"` for K♥
etc.

for a PC / windows machine – use `"cards\\backapcs.gif"`

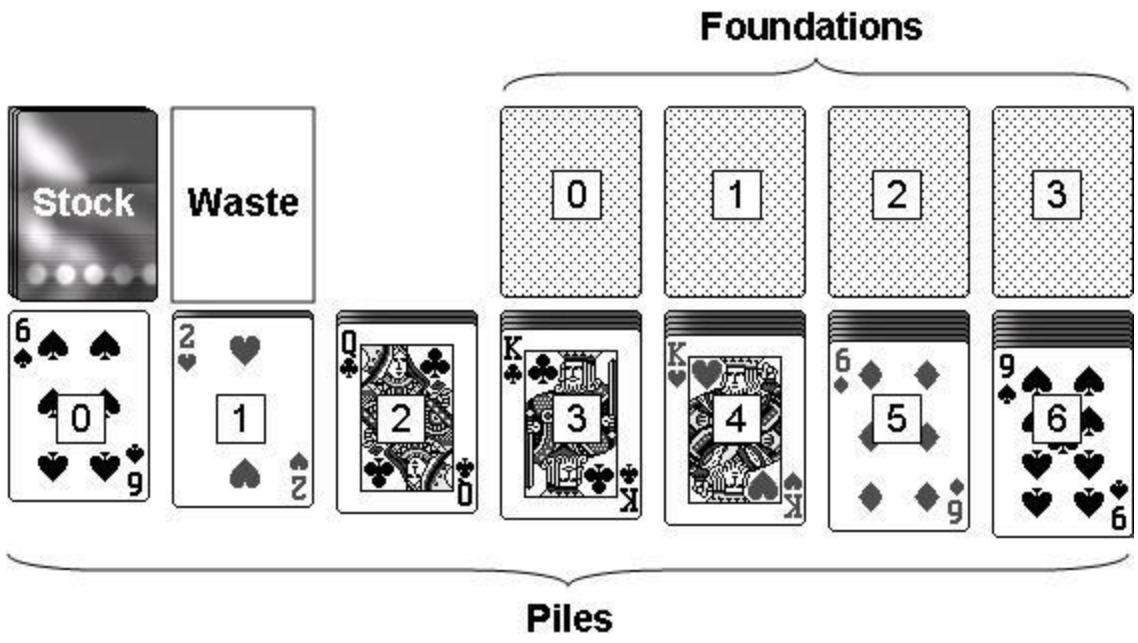
and

`"cards\\ac.gif"` for A♣
`"cards\\2d.gif"` for 2♦
`"cards\\9h.gif"` for 9♥
`"cards\\ts.gif"` for 10♠
`"cards\\jc.gif"` for J♣
`"cards\\qd.gif"` for Q♦
`"cards\\kh.gif"` for K♥

Note: If the cards still do not render on the screen, use the finder or explorer to locate the files and copy the entire path instead of just `cards` folder.

Exercise 2: The `Solitaire` Class

Our solitaire game will be arranged as shown below. Cards from the *stock* are turned face-up onto the *waste*, which may then be moved onto *piles* 0-6. The goal is to get all the cards up onto *foundations* 0-3. We will use this terminology throughout the lab.



The remainder of the code you'll be writing will be in the `Solitaire` class. Open up the `Solitaire` class. Notice that the stock, waste, foundations, and piles are all represented as `Stack`s of `Card`s associated with instance variables. A description of the `Stack` class is shown below.

```
class java.util.Stack<E>
      E push(E item)          // pushes item onto the top of the stack; returns item
      E pop()                 // removes and returns the element at the top of the stack
      E peek()                // returns the element at the top of the stack;
                              // throws an exception if the stack is empty
   boolean isEmpty()
```

Notice that the `Solitaire` class's `display` instance variable maintains a reference to the `SolitaireDisplay` class used to display the cards. This variable is already initialized for you in the constructor.

Write the `Solitaire` class's constructor, which should initialize the stock, waste, foundations, and piles to be empty `Stack`s of `Card`s. (Notice that the constructor already constructs the arrays using the syntax `new Stack[`*num*`]`, because Java mysteriously refuses to let us construct an array as `new Stack<Card>[`*num*`]`. Because of these lines, you will see a warning message whenever you compile. Nonetheless, *you* should be sure to use the `Stack<Card>` syntax everywhere else in the code.)

Now find and complete the methods `getStockCard`, `getWasteCard`, `getFoundationCard`, and `getPile`. These methods should only return a reference to the top card, and should not alter the stock, waste or foundation. For now, ignore all other methods begging you to implement them.

Your `Solitaire` class should now compile and run successfully, producing a blank solitaire board on your screen.

Exercise 3: The Deck

Write the helper method `createStock`, which should create an `ArrayList` containing each of the 52 cards in a standard deck. Then, repeatedly remove a random card from this `ArrayList` and add it to the stock, until there are no cards left to remove. You will now have a shuffled stock of 52 cards. Call `createStock` from the constructor. Go ahead and test that you see a friendly face on the top of the stock pile when you run your code.

Exercise 4: Dealing

Write the helper method `deal`, which should deal cards from the stock to the 7 piles in the arrangement shown in the picture earlier in the lab. Be sure to turn the top card of each pile face up. Call `deal` from the constructor. Test that you now see an arrangement of cards like the one shown.

Exercise 5: Cycling Through the Stock

Write the helper method `dealThreeCards`, which should move the top three cards from the stock onto the top of the waste. If there are fewer than three cards on the stock, just transfer whatever is left to the waste. Be sure to turn up each of the cards you move onto the waste.

Write the helper method `resetStock`, which should repeatedly move the top card from the waste to the top of the stock, until there are no cards remaining in the waste. Be sure to turn down each of the cards you move to the stock.

Find the method `stockClicked`. This method is called by the `SolitaireDisplay` whenever the user clicks on the stock. Modify `stockClicked` so that it tests if the stock has any cards left. If so, transfer three cards to the waste, and otherwise reset the stock, using the helper methods you just wrote.

Test that you can click on the stock to cycle through the cards. You should see 8 cards in each cycle.

Exercise 6: Selecting the Waste

To move cards, the user will click on the card to move and then click on the destination for the card. The `SolitaireDisplay` class keeps track of which card (if any) is currently selected. You will call the following `SolitaireDisplay` methods in order to access or change the selection.

```
void selectWaste()
void selectPile(int index)
boolean isWasteSelected()
boolean isPileSelected()
int selectedPile()
void unselect()
```

Find the method `wasteClicked`. Modify it so that it selects the waste if it is not empty and neither the waste nor a pile is already selected. Unselect the waste if it is already selected.

Also, modify `stockClicked` so that it does nothing if the waste or a pile is selected.

You should now be able to click on the waste to toggle whether or not a yellow border appears around it.

Exercise 7: Moving Cards to Piles

Find the method `pileClicked`. Modify it so that, whenever the waste is selected, clicking on the pile will move the top card from the waste (whatever it may be) onto the top of the given pile, and unselect the waste. Test that you can move cards from the waste to the piles.

Exercise 8: `canAddToPile`

We will now write the helper method `canAddToPile`, as declared below.

```
//precondition:  0 <= index < 7
//postcondition: Returns true if the given card can be
//               legally moved to the top of the given
//               pile
private boolean canAddToPile(Card card, int index)
```

If the top card in the pile is face up, then only a card of the opposite color and next lower rank may be added. For example, only a 6♥ or 6♦ may be played on the 7♣. Only a king may be added to an empty pile, and no card may be added to a pile whose top card is face down.

When you have written `canAddToPile`, modify `pileClicked` so that it only moves a card from the waste to the given pile if the move is legal. Test that your game only allows legal moves now.

Exercise 9: Selecting Piles

Modify `pileClicked` so that it selects the pile when nothing is selected and the user clicks on a non-empty pile whose top card is face up. Alternatively, unselect the pile when the already-selected pile is clicked. Test that you can select and deselect piles correctly.

Exercise 10: Moving Between Piles

We will now write the helper method `removeFaceUpCards`, as declared below.

```
//precondition:  0 <= index < 7
//postcondition: Removes all face-up cards on the top of
//               the given pile; returns a stack
//               containing these cards
private Stack<Card> removeFaceUpCards(int index)
```

For example, suppose the given pile contains the following cards, where *?* denotes a face-down card.

```
?      ?     7♣     6♦     5♠
↑                          ↑
bottom                     top
```

After a call to `removeFaceUpCards`, only the two face-down cards remain, and the returned stack appears as follows.

```
5♠     6♦     7♣
↑             ↑
bottom        top
```

Notice that the top card in the original pile becomes the bottom card in the returned stack.

Next, we will write the helper method `addToPile`, as declared below.

```
//precondition:  0 <= index < 7
//postcondition: Removes elements from cards, and adds
//               them to the given pile.
private void addToPile(Stack<Card> cards, int index)
```

Suppose we use `addToPile` to add the cards in the stack shown above to the following pile.

```
?      9♣     8♥
↑             ↑
bottom        top
```

After the call to `addToPile`, the stack of cards should be empty, and the pile should now contain the following cards.

```
?      9♣     8♥     7♣     6♦     5♠
↑                                   ↑
bottom                              top
```

When you have finished writing `removeFaceUpCards` and `addToPile`, modify `pileClicked` so that when a pile is selected and the user clicks on a different pile, your code removes the cards from the selected pile and attempts to move them to the clicked pile. If those cards can *legally* be added to the clicked pile, go ahead and add them, making sure to unselect the selected pile. Otherwise, if the cards cannot legally be added, add them back to the selected pile. Test that you can successfully move cards between piles when valid.

Exercise 11: Turning Up Cards

Modify `pileClicked` so that, when nothing is selected, clicking on a pile whose top card is face down turns up that card. Test that this works correctly.

Exercise 12: Foundations

We will now write the helper method `canAddToFoundation`, as declared below.

```
//precondition:  0 <= index < 4
//postcondition: Returns true if the given card can be
//               legally moved to the top of the given
//               foundation
private boolean canAddToFoundation(Card card, int index)
```

If the foundation is empty, then only an ace may be added to it. Otherwise, we'll examine the card on the top of the foundation. Only a card of the same suit and next higher rank may be added. For example, only the 4♠ may be placed on the 3♠.

When you have written `canAddToFoundation`, find and modify `foundationClicked` so that it tests if the selected card (either the waste card or the top card of the selected pile) can legally be added to the given foundation. If so, move the card and unselect the source.

Test that your code works correctly. Congratulations! You have earned a ***95%*** (assuming your lab is checked off on time).

Additional Credit Suggestions

In no particular order:

- Allow the user to undo a move, possibly all the way back to the beginning of the game.
- Allow the user to move a card from a foundation to a pile.
- Show all three cards moved from the stock to the waste.
- Provide a shortcut for sending all cards to the foundations that can legally be moved there.
- Have your program celebrate when you win.
- Allow the user to start a new game when they win or give up.
- Allow the user to move only some of the cards in a pile to a new pile.
- Allow the user to double-click on a card to move it to a foundation.
- Provide a scoring system.
- Keep track of the time elapsed in the game.
- Write a class that implements the `Queue` interface using `ListNode`s.