First Semester Review Lab

Spider Solitaire Student Guide

Introduction

Objective

The objective of Spider Solitaire is to remove all cards from the board in as few moves as possible.

Setup

Spider Solitaire is played with two decks of standard playing cards. The game can be played with one, two, or four suits depending on the level of difficulty desired. In this lab, we will use only one suit. You may extend this to multiple suits if you would like but it's not required.

The combined deck is shuffled and half the cards are placed as evenly as possible onto 10 stacks at the top of the board. Only the last card in each stack is face up as shown below. The remaining cards are placed face down in a draw pile at the bottom of the board.

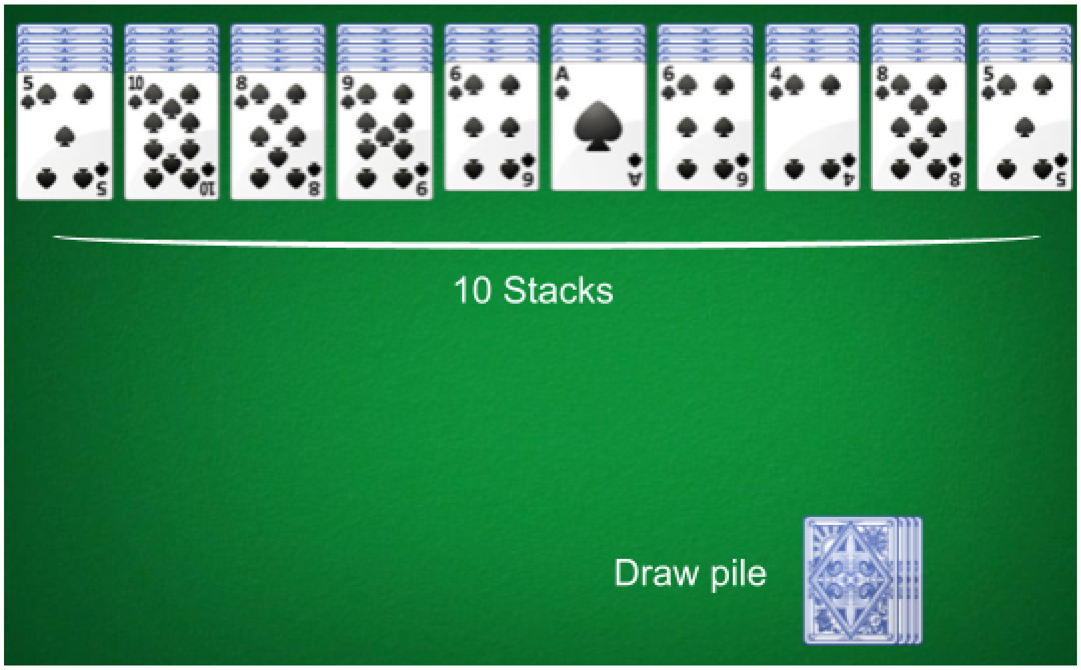


Fig 1.1 - Board setup

How To Play

To win, you must remove all cards from the table by making runs in descending order from King through Ace. If more than one suit is being used then the runs must also be matching suits. Once you have made a run from King to Ace you can remove those cards from the board as long as they appear at the end of a stack.

Valid Moves

1. You may move a run of any length from one stack to another as long as placing the run results in a new, longer run or the destination stack is empty.

2. At any time you may draw from the draw pile. When drawing from the draw pile you must place one card face up at the bottom of each stack. If the draw pile runs out of cards then place as many as will fit starting from the leftmost stack.

3. You may not draw from the draw pile if there are empty stacks on the board. You must fill them with at least one card before drawing.

Game Play Video

If you were absent, here's a video showing how to play the game

Activity 1: Play Spider Solitaire Using 1 Suit

Search the Internet for "Spider Solitaire" and find an online site to play the game. Play a few games using 1 suit until you are able to win at least one game.

Activity 2: Design and Create a Card Class

Introduction

In this activity, you will complete a basic `Card` class that can be used in a variety of games.

Think about card games you've played. What kinds of information do these games require a card object to "know"? What kinds of operations do these games require a card object to provide?

Background

Now think about implementing a class to represent a playing card. What instance variables should it have? What methods should it provide? Discuss your ideas for this `Card` class with classmates.

Download the starter code here and read the partial implementation of the `Card` class. As you read through this class, you will notice the use of the **`@Override`** annotation before the toString method. The Java **`@Override`** annotation can be used to indicate that a method is intended to override a method in a superclass. In this example, the Object class's toString method is being overridden in the `Card` class.

If the indicated method doesn't override a method, then the Java compiler will give an error message. Here's a situation where this facility comes in handy. Programmers new to Java often encounter problems matching headings of overridden methods to the superclass's original method heading. For example, in the Weight class below, the toString method is intended to be invoked when toString is called for a Weight object.

```java
public class Weight {
	private int pounds;
	private int ounces;
	...
	public String tostring(String str) {
		return this.pounds + " lb. " + this.ounces + " oz.";
	}
	...
}
```

Unfortunately, this doesn't work; the tostring method given above has a different name (it's lowercase) and a different signature (it has a parameter) from the Object class's toString method which is spelled as "toString" and has no parameters. The default toString method can only be overridden by having the same method signature as the inherited version found in the Object class. A correctly overridden toString is shown below:

```java
@Override
public String toString() {
	return this.pounds + " lb. " + this.ounces + " oz.";
}
```

The `@Override` annotation will alert the programmer (via compiler error) that the method does not properly override a parent method.

Therefore, whenever you override a method in a subclass use the `@Override` annotation to ensure that you have typed the method signature correctly and are actually overriding an inherited method.

```java
public class Weight {
	private int pounds;
	private int ounces;
	...
	@Override
	public String toString() {
		return this.pounds + " lb. " + this.ounces + " oz.";
	}
	...
}
```

Exercises

Download the Activity 2 Starter Code and complete the following steps in the `Card` class:

1. Read through the starter code to understand what's already there
2. Complete the Constructor so it initializes the class attributes
3. Complete the getSymbol() method
4. Add a getter method to access the Card's value

5. Complete the equals() method. This method should return whether this Card is equal in value to the Card being passed in as a parameter.
6. At the top of the class you'll notice that Card implements an interface:

    ```
    public class Card implements Comparable<Card>
    ```

    As a refresher of Lesson 11, an interface is like a class but consists only of method declarations. For example, here is an interface that describes the methods required for an object to be "Openable."

    ```
    // Openable.java

    public interface Openable {

          public boolean isOpen();
          public void doOpen();
          public void doClose();
    }
    ```

    A class using this interface would declare that it implements the interface and then that class would need to override each method in the interface.

    ```
    // Window.java

    public class Window implements Openable {

          public boolean isOpen() {
                // Some code here
          {
          public void doOpen() {
                // Some code here
          }
          public void doClose() {
                // Some code here
          }
    }
    ```

    In this assignment, we have the Card class implement Java's built-in Comparable<E> interface:

    ```
    public class Card implements Comparable<Card>
    ```

    This means that the Card class must include specific methods listed in the `Comparable` interface. Look up the Java `Comparable` API online and see what methods you must include to satisfy the interface. Complete this method in your Card class. The required method should be declared as `public` and it should return the difference in value between this Card and the Card being passed in as a parameter.
7. Complete the toString() method
8. Add Javadoc comments for all the methods that don't yet have a Javadoc comment
9. Write code in the CardTester class to create several cards and test each method of the Card class.

Activity 3: Design and Create a Deck Class

Introduction

Think about a deck of cards. How would you describe a deck of cards? When you play card games, what kinds of operations do these games require a deck to provide? When you think about the operations, don't include specific game rules. Designing a Deck is about writing a general Deck class that would apply to a wide variety of games. A Deck should only keep track of Cards using some data structure and provide methods that only a Deck (not a game) should be able to perform. For example, `shuffle()` is a standard Deck operation.

Background

Now consider implementing a class to represent a deck of Cards. How will your Deck keep track of Cards? What other instance variables will you need? What methods should a Deck provide?

Exercises

1. Since a Deck keeps track of a list of Cards, you will need an instance variable that keeps track of the Cards in some sort of data structure. You may use any implementation you'd like but think about the advantages and disadvantages before you commit. What are the advantages or disadvantages to using an ArrayList of Card objects versus a regular array of Card objects? If you'd like, you can even use another data structure such as a Stack of Card objects or a LinkedList of Card objects if you think it's appropriate.

   Make a decision and add an attribute to your Deck class to keep track of the Cards in the Deck.

2. Add a constructor to your Deck class that initializes the Deck to an empty state. It may be tempting to add initial Cards to the deck in your constructor but since a Deck may contain any type or number of Cards for different card games, it's best not to add Card objects in the constructor. Let the user that creates the Deck decide what to add.

3. Add methods for all the things you think a Deck should provide. Imagine there was a Deck API you wanted to use to make a card game. What methods would you want in that API? Among these methods, make sure you have a `shuffle()` method and an overridden `toString()` method that uses the **`@Override`** annotation.

   Note: You must write the `shuffle()` method yourself and not rely on any built-in Java functions that do the shuffling for you. In order to shuffle the Deck, here is one possible algorithm:

```
Loop through each index position 'i' in the Deck
    Swap the Card at position i with another Card at a
```

```
                random position within the Deck
```

4. Earlier we said that the Deck should not fill itself with default Cards. The DeckTester is the appropriate place to build a set of Cards and add them to the Deck in order to get the exact Deck you want for a particular game. Suppose your Card class only keeps track of symbols (Ex: "K", "9", "A", etc) and values (Ex: Ace is worth 1, King is worth 13). In such a case, here is a clever way to construct all the Cards and add them to the Deck :

```
String[] symbols = {"A", "2", "3", "4", "5", "6", "7", "8", "9", "T", "J", "Q", "K"};
int[] values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

loop over the index positions 'i' of symbols
  deck.add(new Card(symbols[i], values[i]))
```

   If your Card objects include suits or other attributes like color then you would simply add more nested for-loops to build all the possibilities:

```
String[] suits = {"Apple", "Banana", "Cherry"};

loop over the index positions 'i' of symbols
  loop over the index positions 'j' of suits
    deck.add(new Card(symbols[i], values[i], suits[j]))
```

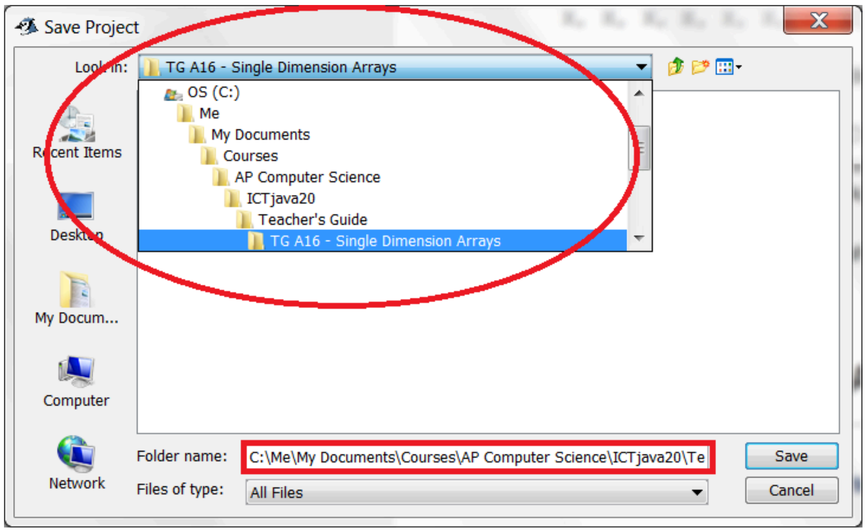
5. Write code in the DeckTester class to create a Deck of Cards and <u>rigorously test all of your methods</u>. It's extremely important that you test your methods thoroughly because you'll soon be relying on them to write the game of Spider Solitaire.

Submission form for Activity 1 through 3

How to Submit - BlueJ users

Rather than submit specific .java files as we have done in the past, for this project you will be uploading your **<u>entire BlueJ project folder</u>** that contains everything. To do this, follow these steps carefully:

1. First you need to know where your BlueJ project folder is located on your computer. To find out, choose Project → SaveAs from the BlueJ menu. To find out where your project is stored on your computer either use the dropdown menu (shown as an oval below) or look at the folder name (shown as a rectangle below). Both of these tell you the path where your BlueJ project is located.



Look in: TG A16 - Single Dimension Arrays

OS (C:)
Me
My Documents
Courses
AP Computer Science
ICTjava20
Teacher's Guide
TG A16 - Single Dimension Arrays

Folder name: C:\Me\My Documents\Courses\AP Computer Science\ICTjava20\Te

Files of type: All Files

2. Open your hard drive in Windows Explorer, Finder, or however you navigate files on your computer and go to your BlueJ project folder.

3. Select **all the files (everything!!)** in your project folder and zip them up into a single compressed file. On most Windows machines you can select all the files, right-click, and choose Send to → Compressed (zipped) folder. Other options are to download and use WinRar or WinZip.

   On most Mac machines you can select all the files, right-click, and choose "Compress Items"

   If you have any problems, search the Internet for "how to create zip files on …" mac or windows

4. Name your compressed file: P#_Last_First_Solitaire_Act_1_to_3.zip

5. Upload your zip file to the submission form

How to Submit - Eclipse, IntelliJ, other users

1. First, make sure your project uses only the default package. You should not see ANY package declarations at the top of any of your classes. Rebuild your project without packages if needed.

2. Find your project folder on your computer and select only the .java source files. Send them to a compressed ZIP file and then upload the ZIP file to the submission form.

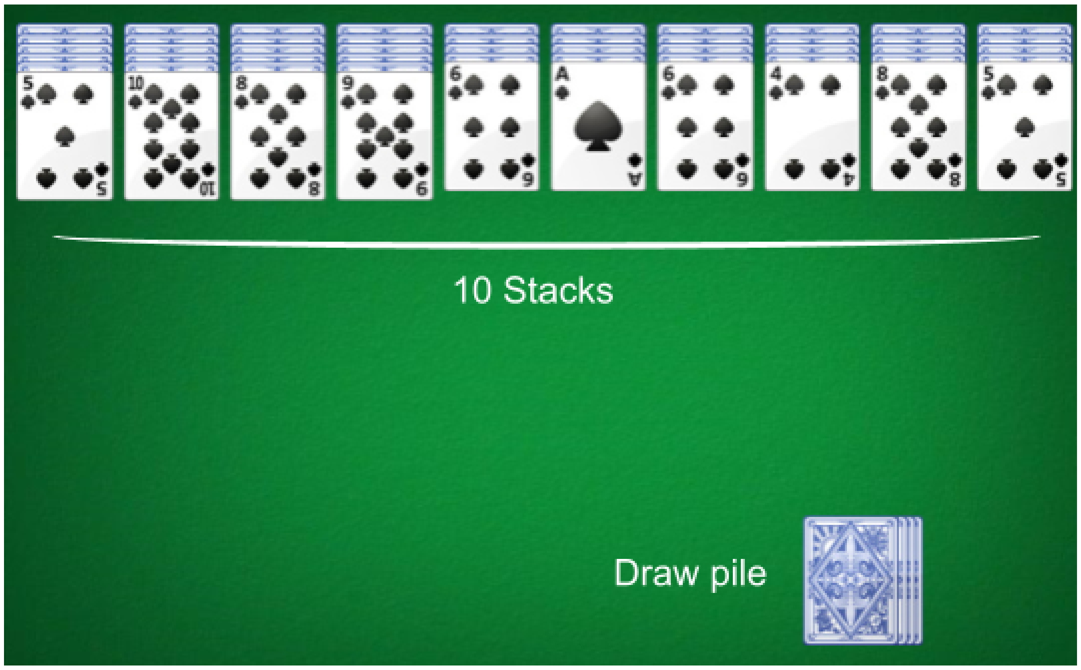
Submission form

Click here for Activity 1 to 3 Submission Form

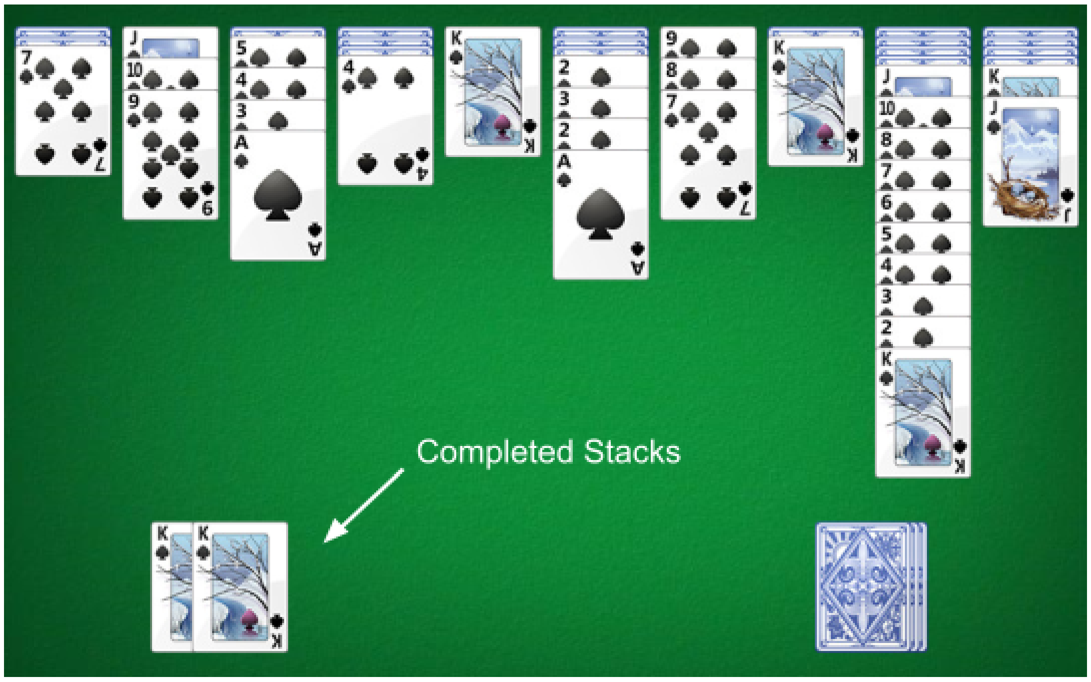
Activity 4: Design and Create a Board Class

Introduction

In the game of Spider Solitaire, the board consists of N "stacks" and a draw pile as shown below.



Most implementations of Spider Solitaire also include a "completed stacks" area where the completed runs are moved to when they are finished. It's your choice whether you want to remove completed runs from the game altogether or move them to a "completed stacks" area.



Background

Now consider implementing a class to represent a Board for the game of Spider Solitaire. All of our code so far has been general and could work with many different card games. The Board class is the first class that is specific to Spider Solitaire. How will your Board keep track of the stacks? How will your Board keep track of the draw pile?

Exercises

1. Decide how your Board will keep track of the stacks and draw pile. Since the number of stacks can vary from game to game, you won't be able to hardcode each stack as its own variable. You will either need to keep an array of Decks or an ArrayList of Decks. After all, a Deck is just a list of Cards. Make your decision and add the necessary class attributes to keep track of the stacks and drawpile (and the completed stacks if you want them).

2. Next, complete the Board constructor which should build a combined Deck of cards consisting of numDecks Decks, shuffle it, and then distribute half the cards to the stacks and the other half to the draw pile. See the constructor comments in the starter code for more detail.

3. Next, complete the printBoard method so that it prints the entire Board as the user should see it. Shown below is one possible format:

   ```
   Stacks:
   1: [X, X, X, 7]
   2: [X, X, X, K]
   3: [X, X, X, J]
   4: [X, X, X, 6]
   5: [X, X, X, 5]
   6: [X, X, 2]
   7: [X, X, 4]

   Draw Pile:
   [X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X, X,
   X, X, X]
   ```

   Notes:
   - `'X'` is the way a Card displays itself if it is face down. Otherwise its symbol is displayed.
   - Half the shuffled cards were distributed to the Stacks. The other half became the Draw pile.

4. Test your code by running the Driver and using the game command "restart" several times to initialize the Board. For debugging it would be a good idea to have all Cards face up. Try creating different Boards by changing NUM_STACKS and NUM_DECKS found in SpiderSolitaire.java.

5. That's all for now. You will finish the rest of the Board class in Activity 5.

Activity 5: Add Game Logic to the Board Class

Introduction

At this point we have a working Card class, Deck class, and a Board that holds Cards in stacks and a draw pile (and a completed stacks area if you have one) and can display itself in the terminal window. The next step is to write the methods that the Board uses to move Cards around and makes sure those moves are valid.

As you complete today's work you should add methods to your Deck that you may need to complete the Board class. For example, you might find that you need a method in your Deck for `Card get(int index)`.

Exercises

1. Complete the `isEmpty` method in the Board class. This method returns true if all stacks and the draw pile are all empty.

2. Temporarily modify your Board constructor to test that your isEmpty method works properly. After initializing and filling the board in different ways, print out whether the Board is empty. You should test at least these cases:
    a. The stacks and draw pile have cards
    b. Some stacks have cards and the draw pile is empty
    c. The stacks are empty but the draw pile has at least one card
    d. The stacks and draw pile are empty

3. Complete the `drawCards` method in the Board class. This method moves one card face up onto each stack from the draw pile. If there aren't enough cards in the draw pile, it moves as many cards as are available.

4. Complete the `clear` method in the Board class. This method removes a run from the end of a given stack as long as the run is complete and appears at the end of the stack. If there is a run of A through K starting at the end of sourceStack then the run is removed from the game or placed into a completed stacks area. If there is not a run of A through K starting at the end of sourceStack then an invalid move message is displayed and the Board is not changed. Or, if the sourceStack value is out of bounds then print an error message and don't change the Board.

5. Temporarily modify your Board constructor to test that your clear method works properly. You'll need to hardcode adding Cards to the stacks to test these cases:
    a. A stack with a complete run of A through K in the correct order and no extra cards. The clear() method should clear these cards.
    b. A stack with a complete run of A through K in the correct order with extra cards at the top of the stack. The clear() method should clear these cards.
    c. A stack with a complete run of A through K in the correct order with extra cards at the bottom of the stack. The clear() method should report an error.

d. A stack with a complete run of A through K in the reverse (wrong) order with no extra cards. The clear() method should report an error.
e. A stack with a partial run of A through K, starting with A but ending early. The clear() method should report an error.
f. A stack with a complete run of A through K in the correct order, but the first part of the run consists of cards that are face down. The clear() method should report an error (you can't clear cards that are face down).

6. The `makeMove` method moves a run of Cards from one stack to another. It begins by searching backward through the specified source stack for a specific Card matching the parameter symbol. If a matching Card is found and the run is valid then the run is moved to the end of the destination stack. If a matching Card is not found, then makeMove prints an error and does not modify the board. Since this method has a lot of game logic to consider, we'll complete it in smaller parts.

   Here is the makeMove method header:

   ```
   /**
   *  Moves a run of cards from src to dest (if possible) and turns the
   *  next card face up (if one is available).  Change the parameter list
   *  to match your implementation of Card if you need to.
   */
   public void makeMove(String symbol, int src, int dest)
   ```

   At first, don't worry about checking for valid runs. Write code in this method so it searches backward starting at the end of the source stack for the given card symbol. If the symbol is found, move all of the cards from the symbol card through the end of the stack onto the end of the destination stack. Then flip the next card face up in the source stack if it is not already face up.

7. Once you have the ability to move a set of cards from one stack to another, it's time to add the game rules. These rules should be integrated into your makeMove method. If the user calls makeMove and the move is invalid, print out an illegal move message. Otherwise, make the move.

   a. A run can only be moved if all the cards from the end of the stack up to the symbol card are in increasing order with no gaps. Here are some examples:

   ```
   Stack 1: [A, T, 9, 8, 7]   // Note: "T" represents a 10 card
   Stack 2: [J, 8, 6, 5, 4]
   Stack 3: [6, 5, 4, Q, J]
   Stack 4: [K, A, 7, 3, 9]

   makeMove("T", 1, 3)   // Legal
   makeMove("8", 1, 4)   // Legal
   makeMove("8", 2, 4)   // Illegal because the run 8, 6, 5, 4 in
                         // stack 2 is incomplete.  It's missing a 7.
   makeMove("6", 3, 1)   // Illegal because the run 6, 5, 4 doesn't
                         // go to the end of the stack
   ```

b. A run can only be moved if all cards in the run are face up. This is a tricky special case where your game might allow a loophole where the player can specify a symbol that is face down and not yet visible. Print an error saying the symbol card is not found in the stack (even though it might be face down in the stack).

```
Stack 1: [X, 5, 4, 3, 2]   // Suppose the hidden card is "6"
Stack 2: [K, A, 5, 3, 7]

makeMove("6", 1, 2)   // Illegal because the "6" is face down
```

c. A run can only be moved if placing it on the destination stack creates a longer, valid run

```
Stack 1: [A, T, 9, 8, 7]   // Note: "T" represents a 10 card
Stack 2: [J, 8, 6, 5, 4]
Stack 3: [6, 5, 4, Q, J]
Stack 4: [K, A, 7, 3, 9]

makeMove("T", 1, 3)   // Legal, creates a valid longer run
makeMove("8", 1, 4)   // Legal, creates a valid longer run
makeMove("9", 1, 3)   // Illegal, new run missing 'T' (ten)
```

8. Modify your `drawCards` methods to match the rule that you can only draw cards from the draw pile if there are no empty stacks.

Submission form for Activity 4 through 5

Zip up your entire project folder as you did for Activities 1 through 3 and submit to the form below.

Submission form

[Click here for Activity 4 to 5 Submission Form]

Activity 6: Error Handling

Introduction

After completing Activities 1 through 5 you should have a working game of text-based Spider Solitaire. The game works smoothly as long as the user enters valid information when inputting commands. But what happens if the user enters something like `>move K 1 six`? Try it. The program will crash because the move command expects two integers to follow "move". The same thing is true for the "clear" command. If the user types `>clear K 1 six` then the nextInt() method will crash the program.

In this Activity you will add try-catch blocks to prevent any chance of the user crashing the program.

Exercises

1. Run the game and try entering an invalid move such as `move K 1 six`. Your program will crash and report the exception that was thrown. Note which line crashed. Your job is to fix this by surrounding the potentially offensive code with a try-catch block. In the catch section, print an error message so the user knows that invalid information was entered.

2. Find all the ways the player of your game can crash your program by entering invalid information. Then add try-catch blocks to prevent these lines from crashing the game and print an error message to the user. Either have the user try again or return them back to the game menu.

3. When this activity is complete, the user should not be able to crash your program in any way.

Submission form

Click here for Activity 6 Submission Form

Activity 7: Saving and Loading Games

Introduction

In this Activity you will add features to save the game at any point and to restore these saved games. Instead of hardcoding a specific filename you will use a JFileChooser dialog window to let the user choose a filename and location to save to. How you save the game information to a file will be for you to decide but this Activity will make the process easier by breaking the problem into smaller parts.

Exercises

1. In your Deck class create a method that returns the complete state of the Deck as a String. The exact format is up to you but it should include all the information necessary to create a copy of this Deck with all the same Cards and their attributes. (i.e. You will need to include all the attributes of each Card in this Deck in order to recreate the same Deck later on.)

2. Test your new method using the DeckTester class by creating a Deck and using your new method to show the full String representation of this Deck. Make sure the String returned contains all the information needed to create the same Deck from scratch.

3. In your Deck class add a constructor that takes a String in the same format as you used in problem 1 and initializes the Deck to that state. Use the DeckTester class to test this method before moving on. Once working, you should be able to create a Deck of Cards from an input String that follows the same format as you used in problem 1.

4. Edit the SpiderSolitaire class to add save and load commands to the menu. Your new menu will look something like this:

```
Commands:
   move [card] [source_stack] [destination_stack]
   draw
   clear [source_stack]
   restart
   save
   load
   quit
>
```

5. Add a method to the Board class to save the game to a text file. In order to save a game you will need to create a `FileWriter` object that writes all of the information necessary to recreate the game state. This means saving the number of stacks, saving each stack, and saving the draw pile. Because each stack and the draw pile are all Decks, you can use the method you wrote in problem 1 to write Strings representing the different Decks on your board.

   In previous labs we have hardcoded the filename as shown below:

```
File apple = new File("apple.txt");
FileWriter banana = new FileWriter(apple);
```

   Instead of hardcoding the filename you should use a `JFileChooser` object to allow the user to type in or select a filename. `JFileChooser` is a Java Swing component which means it is part of the Java Swing library that contains user input controls like buttons, sliders, input boxes, dialog boxes, etc. Read the API for `JFileChooser` or use the code shown below.

```
// Create a JFileChooser that points to the current directory
JFileChooser chooser = new JFileChooser(".");

// Tell the JFileChooser to show a Save dialog window
chooser.showSaveDialog(null);
```

```
// Ask the JFileChooser for the File the user typed in or selected
File apple = chooser.getSelectedFile();

// Create a FileWriter that can write to the selected File
FileWriter banana = new FileWriter(apple);
```

Note 1: If the user presses the Cancel button then the `File` returned by `getSelectedFile()` is `null`. You should handle this in your program so that your program will not crash when the user presses Cancel.

Note 2: If you are using a Macintosh and step 5 `chooser.showSaveDialog(null)` fails to consistently show the dialog window then you will need to use this code instead:

a. First, import these classes:
   ```
   import java.awt.EventQueue;
   import javax.swing.JFileChooser;
   import java.lang.reflect.InvocationTargetException;
   ```

b. Next, surround your file saving code with his code:
   ```
         try {
              EventQueue.invokeAndWait(new Runnable() {
                      @Override
                      public void run() {
                        // Insert JFileChooser and file
                        // saving code here
                      }
                  });
          }
          catch (InterruptedException e) {
             System.out.println("Error: " + e.getMessage());
          }
          catch (InvocationTargetException e) {
             System.out.println("Error: " + e.getMessage());
          }
   ```

6. Add code to your SpiderSolitaire class so that the save command runs the save method in your Board class. Try playing part of a game and then saving the game. Open the save file in a text editor and make sure all the board data has been written to the file in the format you chose.

7. Finally, add a method to the Board class to restore a saved game. This method should prompt the user for a file to load using a `JFileChooser` and then fill the stacks and draw pile with the cards specified in the save game file.

   When using `JFileChooser` you can show an Open dialog window by replacing

   this line: `chooser.showSaveDialog(null);`

with: `chooser.showOpenDialog(null);`

8. Test your new methods by playing a game, saving it, resetting the game, and then loading the saved game. After loading a saved game you should be able to continue from wherever you left off.

Congratulations! You've created a complete text-based version of Spider Solitaire!

Submission form

Click here for Activity 7 Submission form

Activity 8: Optional Features

Introduction

At this point, your game is perfectly playable but is missing additional features. It's up to you what features to add. Here are some ideas:

Feature Ideas

- Add a 'Hint' command that reveals a possible move or says to draw cards if that's the only possibility.

- Add an 'Undo' command. It's up to you how many moves back to allow the user to undo.

- Add an autosave / restore feature so players can quit the game without saving but resume their game upon returning.

- Instead of 1-suit games, add the ability to play 2-suit, 3-suit, or 4-suit games. To do this, you could subclass Card to have more suits and keep the original code the same. You would only need to override some of the Card methods to work differently.

- What other features can you think of ?

Teacher Brainstorm

Should involve
- Scanner
- File reading / writing
- Iterations
- Recursion
- Strings
- arrays or ArrayLists
- Exception throwing and handling
- Libraries and APIs (ex: Random, Point, etc)
- Javadoc commenting
- DrawingTool (?) not necessarily but maybe
- Math operators
- Scope
- Class Design, getters, setters, multiple constructors
- Interfaces
- Inheritance, method overriding, use of super

- Use recursion to play() instead of a loop
- Subclass the no-suit Card to have Cards that have a suit. The changes should be minimal.

Updates Needed
- ~~Limit the Cards to no suit~~ (Done)
- ~~Limit the Stacks to 7 stacks~~ (Seems unnecessary)
- Clearer instructions that explain assumed details picked up by playing the game. For example, you should build one big deck and shuffle it before dealing cards to the stacks. Might make a good journal question: Bob wants to put A-K into each stack and then shuffle the stack. What do you think?
- ~~When cards are distributed to the stacks, they are distributed one by one to each stack. Add a diagram showing this.~~ (Done via PearDeck questions)
- ~~Add some questions about the choice of ArrayList<Cards> drawPile versus Deck drawPile in a journal question.~~ (Done via PearDeck discussion)
- Setting up the Board tester code takes a lot of work, so provide some sample code for testing the Board methods.