100/100

```
/**
 * Driver module for Bulgarian Solitarire game
 *
 * @author CS 140 Instructors
 * @version 3/27/2017
 */
import java.util.Scanner;
import java.io.*;

public class Project6 {
      public static void main(String args[]) throws IOException
      {
          PrintWriter out = new PrintWriter(new FileWriter("Project6_Output.txt"
));
          Scanner console = new Scanner(System.in);

          // Set up the game.  This method creates a random number of piles
          // in the range 4-8 and puts random number of cards in each pile and
          // makes sure that the total number of cards in all the piles is 45.
          BulgarianSolitaire game = new BulgarianSolitaire();

          game.play(out);   // play the game recording each move in an output fi
le

          out.close();   // do not forget to close the output file

          System.out.println("Output is written to Project6_Output.txt file");
          System.out.println("Good bye!");
      }
}
```

```
import java.io.*;
import java.util.Scanner;
import java.util.Random;
import java.util.Arrays;

/**
 * Represents a model of the Bulgarian Solitaire game
 *
 * @author Phil Fevry
 * @version 1.1
 */
public class BulgarianSolitaire
{
    final public static int DEBUG_MODE         = 0;     // Prints various debug
messages to System.out [0: off | 1: on]
    final public static int DEBUG_BREAKPOINT   = 50;    // Point in cycle where
an infinite loop is probably occuring

    final public static int NUMBER_OF_CARDS    = 45;    // Number of cards in de
ck
    final public static int MIN_PILE_COUNT     = 4;     // Minimum piles
    final public static int MAX_PILE_COUNT     = 8;     // Maximum piles
    final public static int FINAL_PILE_COUNT   = 9;     // Game winning number o
f piles

    protected static int cardsInHand;
    private static int [] piles, uniqueNumbers;
    private boolean firstRun, gameOver;
    private PrintWriter fileOut;

    /**
    * Constructor for BulgarianSolitaire
    */
    public BulgarianSolitaire()
    {
        firstRun        = true;
        piles           = new int[0];
        uniqueNumbers   = new int[0];
        cardsInHand     = 0;

        log("Initialization Complete!");
    }

    /**
    * Contains the main loop for which the game is played.
    *
    * @param     out     Writes outputs to a file
    */
    public void play(PrintWriter out) {
        fileOut = out; // Assingn instance of parameter to global variable.

        if (firstRun) {
            output("Game Begins!\n");

            // Initialize the piles
            CardSplitter.makeInitialPiles();
            output("There are initially " + numberOfPiles() + " piles\n\n");

            // Called so a pile which is the only one with a unique number of ca
rds
            // within a limit gets put into an array so it doesn’t get touched.
            checkForUniqueNumbers();
            printPiles();

            firstRun = false;
        }

        int cycle = 0;
        while (gameNotOver()) {
            checkForUniqueNumbers();
            if (cycle >= DEBUG_BREAKPOINT) {
                output("[Failure]: Detected a probable infinite loop.\n");
                break;
            }
            cycle ++;
            log("****************[Cycle " + cycle + "]****************");
        }

        // Game is over at this point!
        printPiles();
        output("Game is over. It took " + cycle + " tries.\n\n");

    }

    // UTILITY METHODS

    // + Mutators
    /**
    *
    * Picks up a card from each pile and puts them down depending on certain con
ditions.
    *
    */
    private void shiftCards() {
        Arrays.sort(piles);

        // Take one card from each pile over the limit
        for (int i = 0; i < numberOfPiles(); i ++) {

            // Don’t pick up the card if its the last one of its number
            if (isExclusiveUniqueNumber(piles[i])) continue;

            pickupFromPile(i, 1);
            log ("Picked up a card from the pile at index: " + i + " (" + cardsI
nHand + " in hand)");
        }

        // Increase the length of piles[] if were still under the final pile cou
nt and add a card to it
        if (cardsInHand > 0 && numberOfPiles() < FINAL_PILE_COUNT) {
            formNewPile();
        }

        // Now its time to put the cards we have in hand into a pile.
        int index = 0;
        while (cardsInHand > 0) {
           if (index > numberOfPiles()-1) index = 0;
            // Don’t put the card down if the pile has the last of it’s number
            if (isExclusiveUniqueNumber(piles[index])) {
                index ++;
                continue;
            }
            placeInPile(index, 1);
            index ++;
            log ("Placed a card down in the pile at index: " + index + " (" + ca
rdsInHand + " in hand)");
        }
    }

    /**
    *
    * Creates a new pile and places one card from the cards being held in hand i
n it.
    *
    */
    private void formNewPile() {
        piles = Arrays.copyOf(piles, numberOfPiles()+1);
        placeInPile(numberOfPiles()-1, 1);
    }
    /**
    *
    * Adds a unique number found in a deck to the uniqueNumbers array.
    *
    * @param    num     number to add
    */
    private static void addToUniqueNumberArray(int num) {
        uniqueNumbers = Arrays.copyOf(uniqueNumbers, uniqueNumbers.length + 1);
        uniqueNumbers[uniqueNumbers.length - 1] = num;
    }
    /**
    *
    * Takes a specific number of cards from whats in hand and puts it into a spe
cific pile.
    * @param    pile    the pile being modified
    * @param    amt     the amount of cards to put down
    */
    private static void placeInPile(int pile, int amt) {
        cardsInHand -= amt;
        piles[pile] += amt;
    }
    /**
    *
    * Takes a specific number of cards from whats in the pile and puts it in the
 hand.
    * @param    pile    the pile being modified
    * @param    amt     the amount of cards to pickup
    */
    private static void pickupFromPile(int pile, int amt) {
        piles[pile] -= amt;
        cardsInHand += amt;
    }

    // + Accessors
    /**
    * Returns the number current number of piles
    *
    * @return   number of piles
    */
    private static int numberOfPiles() { return piles.length; }
    /*
    * Returns the number of cards in a pike
    *
    * @param    column  the pile being examined
    * @return   number of cards in pile
    */
    private static int cardsInPile(int column) { return piles[column]; }
    /**
    *
    * Checks for unique numbers up to a specific boundary (FINAL_PILE_COUNT).
    * If one is found the method puts it into the uniqueNumbers array. This
    * exists to prevent the game from picking up and putting down cards infinite
ly.
    *
    */
    private void checkForUniqueNumbers() {
        for (int i = 0; i < numberOfPiles(); i ++) {
          // check if within range
          if (piles[i] <= FINAL_PILE_COUNT && piles[i] >= 1) {
              if (uniqueNumberFound(piles[i])) {
                 addToUniqueNumberArray(piles[i]);
             }
          }
        }
    }

    // + Methods that determine if the game is over.
    /**
    * Checks to see if the current number of piles equals the FINAL_PILE_COUNT
    *
    * @return   condition of final pile count
    *
    */
    private boolean validPileCount() {
        if (numberOfPiles() == FINAL_PILE_COUNT) {
            log ("*Final pile condition met*");
            return true;
        }
        return false;
    }
    /**
    *
    * Checks to see if numbers 1 through FINAL_PILE_COUNT exist in a pile
    *
    * @return   condition that piles are made up of all unique numbers
    *
    */
    private boolean allUniqueNumbersFound() {
        for (int i = 0; i < FINAL_PILE_COUNT; i ++) {
            if (uniqueNumberFound(i)) return false;
        }
        log ("*Unique numbers condition met*");
        return true;
    }
    /**
    *
    * Calls allConditionsMet() to see if game is over.
    * If game is not over, shiftCards() method is called to advance the game
    *
    * @return   required conditions not met
    *
    */
    private boolean gameNotOver() {
        if (allConditionsMet()) return false;

        log ("Piles before shift");
        printPiles();

        // Shift cards until game is over.
        shiftCards();

        log ("Piles after shift");
        if (DEBUG_MODE == 1) {
            printPiles();
            printUniqueNumbers();
        }
        return true;
    }
    /**
    * Checks to see if conditions required for the game to be over are met.
    * @return   all conditions are met
    *
    */
    private boolean allConditionsMet() {
        return (validPileCount() && allUniqueNumbersFound());
    }

    // + Methods that deal with unique numbers
    /**
    * Checks to see if a particular number exists in the unique number array
    * @param    num     number to check
    * @return   not in the unique number array
    *
    */
    private static boolean uniqueNumberFound(int num) {
        if (num == 0) return false;
        for (int index: uniqueNumbers) {
            if (index == num) return false;
        }
        return true;
    }
    /**
    * Checks to see if a number is the only one that exists in all piles.
    * Only applies to numbers under or equal to the FINAL_PILE_COUNT.
    *
    * @param    num  number to check
    * @return   number is exclusive
    */
    private boolean isExclusiveUniqueNumber(int num) {
        int count = 0;
        for (int i = 0; i < numberOfPiles(); i ++) {
            if (piles[i] == num && num <= FINAL_PILE_COUNT)
            {
                if (num == 0) break;
                count ++;
            }
        }
        if (count == 1) log("The last " + num + " is here. Start next cycle");
        return (count == 1);
    }
    /**
    * Checks to see if a number exists in the uniqueNumbers array.
    * @param    num  number being checked
    * @return   existence of number
    */
    private boolean uniqueNumberInList(int num) {
        return !uniqueNumberFound(num);
    }
    /**
    * Gets the highest unique number in the uniqueNumbers array.
    * @return   number
    */
    private int getHighestUniqueNumber() {
        int highest = 0;
        for (int i = FINAL_PILE_COUNT; i > 0; i --) {
            if (i > highest && uniqueNumberInList(i))
                highest = i;
        }
        return highest;
    }
    /**
    * Gets the lowest unique number in the uniqueNumbers array.
    * @return   number
    */
    private int getLowestUniqueNumber() {
        int lowest = FINAL_PILE_COUNT;
        for (int i = 0; i < FINAL_PILE_COUNT; i ++) {

            if (i < lowest && uniqueNumberInList(i)) { lowest = i; }
        }
        return lowest;
    }

    // + Methods that print
    /**
    * Prints to both standard output and PrintWriter instance defined in the par
ameter for the play() methods.
    * @param    text    string of text to print
    */
    private void output(String text) {
        System.out.print(text);
        fileOut.print(text);
    }
    /**
    * Calls output() method to print the current number of cards in each pile.
    */
    private void printPiles() {
        for (int column: piles) {
            output(column + "   ");
        }
        output("\n\n");
    }

    // + Debug Methods
    /**
    * Debug method which prints a message starting with "[Debug]: " to standard
output
    * @param    message     text to output
    */
    private static void log(String message) {
        if (DEBUG_MODE == 1)
            System.out.println("[Debug]: " + message);
    }
    /**
    * Debug method which prints the uniqueNumbers array and its boundaries to th
e standard output
    */
    private void printUniqueNumbers() {
        System.out.print("Unique numbers in range: ");

        for (int index: uniqueNumbers)
            System.out.print(index + "   ");

        System.out.print("[Highest is " + getHighestUniqueNumber() + " Lowest is
 " + getLowestUniqueNumber() + "]");
        System.out.print("\n");
    }
    // SUBCLASSES

    /**
    * Represents a CardSplitter who sets up the game.
    *
    * @author Phil Fevry
    * @version 1
    */
    protected static class CardSplitter extends BulgarianSolitaire {
        private static Random randomizer = new Random();
        /**
        * Picks up NUMBER_OF_CARDS and splits them into a random number of piles
.
        */
        private static void makeInitialPiles() {
            // Set the amount of cards in hand
            cardsInHand = NUMBER_OF_CARDS;

            // Generate a random amount of piles
            piles = new int [MIN_PILE_COUNT + randomizer.nextInt(MAX_PILE_COUNT-
MIN_PILE_COUNT)];

            splitCardsInHandIntoPiles();
        }
        /**
        * Puts a random amount of cards down in each pile.
        */
        protected static void splitCardsInHandIntoPiles() {
            for (int i = 0; i < numberOfPiles(); i ++) {

                // Distribution limits
                int min = 1; int max = cardsInHand;

                // In the initial loop, don't put too much in one pile
                if (cardsInPile(i) == 0) max /= numberOfPiles();

                // Randomize amount to put into each pile
                int amount = getRandomNumber(min, max);

                placeInPile(i, amount);
            }

            // Do it again if there are still cards in hand after the last cycle
            if (cardsInHand > 0) splitCardsInHandIntoPiles();
        }
        /**
        * Random number generator
        * @param    min     minimum number
        * @param    max     maximum number
        * @return   a random number between min and max
        */
        private static int getRandomNumber(int min, int max) {
            if (min > max) return 0;
            return randomizer.nextInt(max) + min;
        }
    }
}
```

Discussion Log
Assignment: Project 6
Name: Phil Fevry

Time taken: About a week @ around 5 hours a day each (~35 hours)

What I learned:
- Increased understanding of why its important to split tasks up into various me
thods for readability and ease of modification
- I employed valuable practice on designing and laying out classes.
- Nothing special is needed to really create files. PrintWriter outputs to a fil
e just like it outputs to screen.
- (Offtopic) started using a third-party text-editor with the java compiler inst
ead of BlueJ and I realized the production gains of using a text editor over an
IDE for simple projects.

Difficulties Faced:
-Figuring out how to design the main algorithm
-- The first commit had an infinite loop some but not all the times when it firs
t ran

Resources Used:
Java API

==================
(NEW)
Version 1 changes:
- Refactored and reorganized a lot of code to be less redundent and easier to re
ad
- Made comments more clear and easier to understand
- Sorted the cards before each shiftCards() cycle
- Added JavaDocs

Time taken: 4 hours

What I learned:
- Increased appreciation for making debug/testing tools
- More about Git and importance of version control

Difficulties I faced:
- Figuring out how to fix the infinite loop condition in the last commit

How I fixed it:
- Noticed a pattern with every output where an infinite loop occured

Output in Terminal =
```
[ 1   2   3   4   5   5   8   8   9 ] - 5 & 8
[ 1   2   2   3   5   7   8   8   9 ] - 2 & 8
[ 1   2   3   4   5   5   8   8   9 ] - 5 & 8
[ 1   2   3   3   5   6   8   8   9 ] - 3 & 8
[ 1   2   3   4   4   6   8   8   9 ] - 4 & 8
```

- The number 8 was always involved.
- I figured it must have been an off by one error because 8 is one less than 9
- I tweaked the numbers in shiftCards() and most of the methods that dealt with
unique numbers

```
commit 206c607539a017dccd6717089af2f0d742d46fc0
Author: Phil Fevry <phil.fevry@gmail.com>
Date:   Sun Apr 30 23:15:59 2017 -0400

    FINAL FINAL VERSION - Better JavaDoc -

commit d7e5ca079fa9a2061afcb750048b8e52f6b2ed8f
Author: Phil Fevry <phil.fevry@gmail.com>
Date:   Sun Apr 30 21:59:22 2017 -0400

    Final version Fixed infinite loop issue and refactored a lot of code.

commit c5e090764218c599e08665079352c95cc8ce2dd7
Author: Phil Fevry <phil.fevry@gmail.com>
Date:   Sat Apr 29 22:25:04 2017 -0400

    first commit

commit e5cd83335c381432710e5c53bad8cb4570bedc38
Author: Aparna Mahadev <amahadev@worcester.edu>
Date:   Sun Apr 2 08:31:36 2017 -0400

    Project6
```