CS 455 Programming Assignment 2

Spring 2015 [Bono]
Due: Wed, Feb 25, 11:59pm

Introduction

In this assignment you will get practice working with Java arrays, and more practice implementing your own classes. Like you did in assignment 1 and lab 4, you will be implementing a class whose specification we have given you, in this case a class called `SolitaireBoard`, to represent the board configuration for a specific type of solitaire game described further below. You will also be using tools to help develop correct code, such as `assert` statements along with code to verify that your class is consistent.

Note: this program is due after your midterm exam, but it's a fair amount bigger than the first assignment. We recommend getting started on it before the midterm. It only uses topics from before the midterm, so working on it now will also help you prepare for the exam (there will be paper and pencil array programming problems as part of the exam).

Resources

- Horstmann, Special topic 11.5, `assert` statements
- Horstmann, Section 11.3 Command-line arguments
- Horstmann, Section 8.4 `static` methods
- Horstmann, Section 11.2.5 Scanning a `String`
- Horstmann, Section 7.1.4, 7.3 Partially-filled arrays
- CS 455 lecture, 2/10, Representation invariants
- CS 455 lecture, 2/3, 2/5, 2/10, Partially-filled array example
- Horstmann, Special topic 8.1 Parameter passing

The assignment files

Getting the assignment files. Make a pa2 directory and cd into it. Copy all of the files in the `pa2` directory of the course account to your `pa2` directory. We can accomplish this with Unix wildcards (denoted with a `*`).

```
cp ~csci455/assgts/pa2/* .
```

The files in **`bold`** below are ones you create and/or modify and submit. The ones not in bold are ones that you will use, but not modify. The files are:

- **`SolitaireBoard.java`** The interface for the `SolitaireBoard` class; it contains stub versions of the functions so it will compile. It also contains some named constants. You will be completing the implementation of this class. You may not change the interface for this class, but you may add private instance variables and/or private methods to it
- **`BulgarianSolitaireSimulator.java`** A main program that does a Bulgarian Solitaire Simulation. This simulation is described further in the section on the assignment
- `turninpa2` A shell script with the command for turning in the assignment. See the section on submitting your program for more details.
- **`README`** See section on Submitting your program for what to put in it. Before you start the assignment please read the following statement which you will be "signing" in the README:

  "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people, with the exception of the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

Note on running your program

We will be using `assert` statements in this program. To be able to use them, you need to run the program with asserts enabled (-ea flag). (You do not need to compile any special way.) Here is an example:

```
java -ea BulgarianSolitaireSimulator
```

You should run with this flag set every time.

Assert statements, described in Special Topic 11.5 of the text, are another tool to help us write correct code. More about how we are using them here in the section onrepresentation invariants.

NOTE: In Eclipse you use the Run Configurations settings (in the Run menu) to change what arguments are used when running a program. We will be using both*program arguments* (more details in the next section) and *VM arguments* when running this program; `-ea` is a VM argument.

The assignment

You will be implementing a program to model the game Bulgarian Solitaire. This is problem P7.4 from our textbook Big Java: Early Objects, 5th Edition,

by Cay Horstmann. Here is his description of the problem (second paragraph is a paraphrase):
The game starts with 45 cards. (They need not be playing cards. Unmarked index cards work just as well.) Randomly divide them into some number of piles of random size. For example, you might start with piles of size 20, 5, 1, 9, and 10. In each round you take one card from each pile, forming a new pile with these cards. For example, the starting configuration would be transformed into piles of size 19, 4, 8, 9, and 5. The solitaire is over when the piles have size 1, 2, 3, 4, 5, 6, 7, 8, and 9, in some order. (It can be shown that you always end up with such a configuration.)

In the normal mode of operation your program will produce a random starting configuration and print it. It then keeps applying the solitaire step and printing the results, and stops when the final configuration is reached.

We recommend you finish playing Horstmann's example game to see how it comes out (you can do it with just pencil and paper). (Save your work because we may be asking for it in lab.)

To make it easier to test your code, your program will be able to be run in a few different modes, each of these controlled by a command-line argument. The user may supply one or both of the arguments, or neither.

`-u`
: Prompts for the initial configuration from the user, instead of generating a random configuration.

`-s`
: Stops between every round of the game. The game only continues when the user hits enter (a.k.a., return).

Command-line argument processing is discussed in section 11.3 of the Horstmann text. But to make things a little easier, we wrote the code for processing the command-line arguments for you. It appears in starter code you get in the `main` method in `BulgarianSolitaireSimulator.java`. Here are a few examples of ways to run the program in the Unix shell:

```
java -ea BulgarianSolitaireSimulator -u java -ea
BulgarianSolitaireSimulator -u -s java -ea BulgarianSolitaireSimulator
```

[Note: recall we are using the `-ea` argument for assertion-checking. The arguments after the program name are the ones that get sent to your program.]

There are more details about exactly what your output should look like in each of these operation modes in the section on
the `BulgarianSolitaireSimulator` program.

Some of the requirements for the program relate to efficiency, testing, and style/design, as well as functionality. They are described in detail in the following sections of the document, and then summarized near the end of the document.

SolitaireBoard: interface

What follows is the specification for the SolitaireBoard class. You must implement the following methods so they work as described:

`SolitaireBoard()`
: Creates a solitaire board with a random initial configuration.

`SolitaireBoard(String numberString)`
: Creates a solitaire board with the given configuration. The format of the string is a space-separated list of integers. Example string: "20 5 1 9 10".
Precondition: `SolitaireBoard.isValidConfigString(numberString)` is true
More about this in the section on valid configuration strings.

`void playRound()`
: Plays one round of Bulgarian solitaire. Updates the configuration according to the rules of Bulgarian solitaire: Takes one card from each pile, and puts them all together in a new pile.

`boolean isDone()`
: Returns true iff the current board is at the end of the game. That is, there are `NUM_FINAL_PILES` piles that are of sizes 1, 2, 3, ..., `NUM_FINAL_PILES` in any order.

`String configString()`
: Returns current board configuration as a string with the format of a space-separated list of numbers with *no leading or trailing spaces*. The numbers represent the number of cards in each non-empty pile. Example return string: "20 5 1 9 10"

`static boolean isValidConfigString(String configString)`
: Returns true iff `configString` is a space-separated list of numbers that represents a valid Bulgarian solitaire board, assuming the card total`SolitaireBoard.CARD_TOTAL`. Note this is a static method: it only operates on its explicit `String` argument; no implicit object instance involved. More about this in the following section.
Precondition: `configString` must only contain numbers and whitespace

In addition, we have defined two public named constants for you:

`static final int CARD_TOTAL`
`static final int NUM_FINAL_PILES`
: We have defined two public named constants for you, `CARD_TOTAL` (45) and `NUM_FINAL_PILES` (9). Please write all of your code in terms of these

constants, so that your program will still work if their values are changed. You should also test your code with different values for NUM_FINAL_PILES. For games that will terminate, the sum of the numbers from 1 to `NUM_FINAL_PILES` must equal `CARD_TOTAL`, so we have set `CARD_TOTAL` to be a value computed from`NUM_FINAL_PILES`. So, for example, if you change `NUM_FINAL_PILES` to 4, `CARD_TOTAL` will automatically have the value 10. See comments in`SolitaireBoard.java` for more details.

Note: No `SolitaireBoard` methods do any I/O.

You may have noticed that there is another method, `isValidSolitaireBoard`, that's `private` , i.e., not part of the public interface. We will describe that later after we discuss the representation.

You may not change the public interface for this class, with the following exception: you may add a public `SolitaireBoard` `toString` method that you may want to use for debugging purposes. It would be very short and mostly consist of call to the `toString` method(s) of its constituent part(s). That way you can see if you are building your `SolitaireBord` object correctly, even before you implement `configString`. Section 9.5.1 of the textbook has more about writing a `toString` method. Hint: to get a String representation of an array, use `Arrays.toString`

More about `isValidConfigString`

So, why does the String version of the SolitaireBoard constructor have a precondition, when the code to check that precondition is part of the class itself? It's because constructors have no return values, so we have no way to return an error code if the client gives invalid data to the constructor. There is another way to get around this, namely Java Exceptions, but we won't get to them until later in the semester. So, instead, your client code (i.e., in `BulgarianSolitaireSimulator`) can call`isValidConfigString` to check whether the string is valid before you create a SolitaireBoard with that data. We'll discuss this method further in a later section; but it won't make sense until we discuss the representation we're using.

SolitaireBoard: representation/implementation

For the purposes of this assignment you are required to use an array to represent the piles of cards in your solitaire board (each array element is one pile). **You may have additional fields, but you may not use an `ArrayList`.** This is going to be a partially-filled array. However, different from other

situations where the number of elements in an array can change size, in this application there is an upper bound on the number of elements, so if you allocate your array in the constructor using that upper bound, you'll never have to resize it later. You'll figure out what that upper bound is once you think about the problem a little.

Another requirement is that you don't create a new array every time you play one round of the game, but rather you will update values in the the same array to do a round. (Put another way, you will only call `new` *once* for the array of piles in a SolitaireBoard object.)

Representation invariants

Many of the development techniques we discuss in this class, for example, incremental development, the use of good variable names, and unit-testing, are to help enable us to write correct code (and make it easier to enhance that code later). Another way to ensure correct code within a class is to make explicit any restrictions on what values are allowed to be in a private instance variable, and any restrictions on relationships between values in different instances variables in our object. Or put another way, making sure we know what must be true about our object representation when it is in a valid state. These are called *representation invariants*.

Representation invariants are things that are true about the object as viewed by the implementor. Since for many classes, once a constructor has been called the other methods can be called in any order, we need to ensure that none of the constructors or mutators can leave the object in an invalid state. It will be easier to do that if we know what those assumptions are.

There are two assignment requirements for your SolitaireBoard class related to this issue (detailed explanations of each of these follow):

1. in a comment just above or below your private instance variable definitions for SolitaireBoard, list the representation invariants for the object.
2. write the private `boolean` method `isValidSolitaireBoardl()` and call it from other places in your program as described below.

The representation invariant comment for `SolitaireBoard`

Write a list of all the conditions that the internals of a SolitaireBoard object must satisfy. That is, conditions that are always true about the data in a valid SolitaireBoard object. For example, one or more invariants would describe

where the data is in a partially filled array (we did a similar example in lecture on 2/10). Another one (or more) would be related to the restriction that there are always `CARD_TOTAL` cards on the board.

`isValidSolitaireBoard()` method
This private method will test the representation invariant for the internals of a solitaire board. It will return true iff it is valid, i.e., the invariants are satisfied.

Call this function at the end of every public SolitaireBoard method, including the constructors, to make sure the method leaves the board in the correct state. This is one kind of sanity check: one part of a program double-checking that another part is doing the right thing (similar to printing expected results and actual results). Note: this only applies to non-static methods. SolitaireBoard also has one public static method -- you won't be able to call isValidSolitaireBoard there since there is no object to check.

Rather than putting this test in an if statement, we're going to put it in an `assert` statement. For example:

```
assert isValidSolitaireBoard();      // calls isSolitaireBoard on
implicit parameter
```

Assert statements are described in Special topic 11.5 of the text.

Please make sure you are running your program with assertions enabled for every run of this program, since it's in a development stage. See earlier section for how to do this. You won't really know if they are getting checked unless you force one to fail.

The point of these assert statements is to notify you in no uncertain terms of possible bugs in your code. The program crashing will force you to fix those bugs. For example, if a board doesn't have `CARD_TOTAL` (45) cards on it, then the simulation may never terminate, or if the array has "hole" in the middle, then other methods, such as `configString` may not work as advertised.

Relationship between `isValidSolitaireBoard` method and `isValidConfigString` method

So, we've determined a client (e.g., `BulgarianSolitaireSimulator` code) can check whether their String represents a valid solitaire configuration by calling the public static method `isValidConfigString` (that was described further here). And we've said that the implementation of `SolitaireBoard` will need to call the private method `isValidSolitaireBoard` to verify that our class does not violate

our representation invariants when we call methods on it (that was described further in the previous section). But it seems like these two methods are doing the same checks, right? Yes, pretty much.

Furthermore, we can structure our code so we don't have to write the code for this check more than once. We'll give you some hints on how to do so here; it's a little confusing, because it involves both static and non-static methods. **You are not required to employ the code reuse here** but I would expect more advanced students to do so. In fact, you may want to skip this section for now, and come back to it once you are well into your design and coding for the assignment.

So, to continue, to reuse the code to check if it's a valid configuration involves factoring out the common code into a third method, like we did when we created `lookupLoc` to help us write `insert` and `remove` for the `Names` class we developed in recent lectures.

But because one of the methods here is static, it means this third method will also need to be static. It would take a partially-filled array version of the configuration as its two parameters; it might have the following header:

```
private static boolean isValidConfiguration(int[] piles, int numPiles)
```

So, the non-static method `isValidSolitaireBoard` would just call this new method, using its instance variables as the actual parameters in the call to isValidConfiguration. Since isValidConfiguration is static, it can't access the instance variables directly.

And the static method `isValidConfigString` would convert its string to a partially-filled array of integers version, and then call `isValidConfiguration` to do the actual check.

A small downside to this organization is that at run time the string will end up getting converted to the array representation twice, once when the client checks if its a valid representation, and then again in the SolitaireBoard constructor. But you don't have to *write* that conversion code twice: you can do the same trick again: make another private static method to do the conversion, and then call it in both places. (Hint: to return two values from a Java method as would be necessary here, one of them can be the actual return value, and the other can be an updated array. This works because array parameters (like objects) are passed as references. (See Special topic 8.1 for more on Java parameter passing.))

`BulgarianSolitaireSimulator` program

Please take a look at this example for what your output must look like. This shows (part of) a run of the program with the **`-u`** option turned on. It also illustrates the error-checking. User input is shown in bold.

```
Number of total cards is 45
You will be entering the initial configuration of the cards (i.e., how
many in each pile).
Please enter a space-separated list of positive integers followed by
newline:
40 1 1 1 1
ERROR: Each pile must have at least one card and the total number of
cards must be 45
Please enter a space-separated list of positive integers followed by
newline:
100 -55
ERROR: Each pile must have at least one card and the total number of
cards must be 45
Please enter a space-separated list of positive integers followed by
newline:
0 45
ERROR: Each pile must have at least one card and the total number of
cards must be 45
Please enter a space-separated list of positive integers followed by
newline:
40 1 1 1 1 1
Initial configuration: 40 1 1 1 1 1
[1] Current configuration: 39 6
[2] Current configuration: 38 5 2
[3] Current configuration: 37 4 1 3 . . .
[30] Current configuration: 10 2 3 4 5 6 7 8
[31] Current configuration: 9 1 2 3 4 5 6 7 8 Done!
```

Here is an example of what your output should look like with the **`-s`** option turned on (the -u option is not set in this example, so it uses a random initial configuration). Again, only part of the run is shown here. After each "<Type return...>" the program blocks until the user hits the return key.

```
Initial configuration: 9 4 6 26
[1] Current configuration: 8 3 5 25 4
<Type return to continue>
[2] Current configuration: 7 2 4 24 3 5
<Type return to continue>
[3] Current configuration: 6 1 3 23 2 4 6
<Type return to continue>
. . .
[26] Current configuration: 2 3 4 5 6 7 8 10
<Type return to continue>
[27] Current configuration: 1 2 3 4 5 6 7 9 8
<Type return to continue>
Done!
```

If neither argument is set, then the program will take no user input, and just show the initial configuration followed by the

A correct program will always terminate. You should leave in the code that we gave you that sets `SolitaireBoard.CARD_TOTAL` in terms of the value of `SolitaireBoard.NUM_FINAL_PILES` (the code to do that is already present in the starter version); this way it should still terminate even if you change `NUM_FINAL_PILES` to some other positive value.

Your output for a particular input should match what's shown above character-by-character (e.g., the messages displayed and the error handling should be the same), so we can automate our tests when we grade your program.

Error checking required

You are required to check that the list of numbers entered is a valid configuration (you can use your method `SolitaireBoard.isValidStringConfig`), but you are not required to check that all the values entered are numeric. Put another way, we will not test your program with non-numeric input. The first example in the previous section shows the error message you should print for an invalid configuration.

Converting a String into an array of numbers

In user mode, you will not be able to read in the numbers directly into an array of ints using repeated calls to `nextInt`, because we're using newline as a delimiter (i.e., a signal that that's the end of the input data), and `nextInt` skips over (as in, doesn't stop for) newlines. Also if you look at the interface for SolitaireBoard, you'll see that there is no constructor to create one from an array of numbers, but there is one that takes a String. So you'll want to read the input all at once using the `ScannernextLine` method, and the conversion to an array of ints will actually happen in the `SolitaireBoard` class. (`SolitaireBoard.isValidConfigString` will also probably need to do this conversion. This section has more about reusing the same code for these in both places.)

How do we do such a conversion? Section 11.2.5 of the textbook (called Scanning a String) shows one way of solving the problem of processing an indeterminate number of values all on one line. It takes advantage of the fact that the `Scanner` class can also be used to read from a `String` instead of the keyboard. Once we have our String of ints from the call to `nextLine()`, you create a *second* `Scanner` object initialized with this string to then break up the line into the parts you want, using the Scanner methods you are already familiar with.

Structure of `BulgarianSolitaireSimulator`

The code for `BulgarianSolitaireSimulator` is too long to be readable if you put it all into the `main` method. We could design and add another class, to deal with the simulation, but instead we'll use a procedural design to organize the code; we'll review procedural design here.

A good design principle (for procedural as well as object-oriented programming) is to keep each of your methods small, for easier program readability. In object-oriented programming, the class design sometimes naturally results in small methods, but sometimes you still need auxiliary private methods. The same principles apply for a procedural design. Since we haven't given you a predefined method decomposition for the `BulgarianSolitaireSimulator`, you will have to create this decomposition yourself.

A procedural design in Java is just implemented as `static` methods in Java that pass data around via explicit parameters. Static methods are discussed in Section 8.4 of the text, and this use of them was also discussed in a sidebar in Lab 4. We have seen several examples of this in other test programs we have

written, for example`NumsTester.java` of lab4 and `PartialNamesTester.java` we developed in lecture. We have also seen some utility classes in Java that have static methods: `Math` and`Arrays`.

If you have learned about procedural design in previous programming classes, you know that global variables are a no-no. Thus, in designing such a "main program" class, we don't create any class-level variables, because they become effectively global variables (see also Style Guideline #9). The "main class" does not represent any overall object. Instead you will create variables *local* to `main` that will get passed to (or returned from) its helper methods, as necessary.

Note: the next section discusses a limit on method length as one of our style guidelines for this course.

Summary of requirements

As on the first assignment, there are several requirements for this assignment related to design, testing, and development process strewn throughout this document. We'll summarize those and the functional requirements here:

- implement SolitaireBoard class according its public interface (see SolitaireBoard: interface)
- use the representation for SolitaireBoard described in the section about that.
- write representation invariant comments for SolitaireBoard class.
- implement and use private SolitaireBoard method `isValidSolitaireBoard` as described here.
- implement BulgarianSolitaireSimulator with the user interface described in the section about the BulgarianSolitaireSimulator program.
- do the error checking described in the Error Checking section.
- your code will also be evaluated on style and documentation. We will deduct points for programs that do not follow the published style guidelines for this course (they are also linked from the Assignments page). (Note: For pa1 we only deducted points for problems related to *some* of the style guidelines.) One guideline we want you to be especially aware of is the limit of 30 lines of code at most allowable in a method. This is exclusive of whitespace, comment lines, and lines that just have a curly bracket by itself (i.e., you should not sacrifice code-readability to make your code fit into this limit). Hopefully, it's obvious that putting multiple statements or declarations on a single line decreases code readability and is not desirable; that would also result a loss in style points.

README file / Submitting your program

You will be submitting `SolitaireBoard.java`, `BulgarianSolitaireSimulator.java`, and `README`. Make sure your name and loginid appear at the start of each file.

Here's a review of what goes in the README: This is the place to document known bugs in your program. That means you should describe thoroughly any test cases that fail for the the program you are submitting. (You do not need to include a history of the bugs you already fixed.) You also use the README to give the grader any other special information, such as if there is some special way to compile or run your program. You will also be signing the certification shown near the top of this document.

Like last time, for your convenience we put the exact submit command to use in a shell script. There should be a copy of this script in your aludra directory that has all your source code files in it (it was one of the original files you copied from our assignment directory), and you can run it as follows:

```
source turninpa2
```
(It's just a text file, so you can look at its contents to see the submit command it will run.)