CS 0445 Spring 2022 Assignment 1

Online: Wednesday, January 19, 2022
Due: All source files plus a completed Assignment Information Sheet zipped into a single .zip file and submitted properly to the submission site by **11:59PM on Saturday, February 5, 2022** (Note: see the submission information page for submission details)
Late Due Date: 11:59PM on Monday, February 7, 2022

Purpose: To refresh your Java programming skills, to emphasize the object-oriented programming approach used in Java and to thoroughly understand the specifications and implementation of the Queue ADT. Specifically, you will work with control structures, class-building, interfaces and generics to **create** and **utilize** a simple array-based (enhanced) Queue.

Goal 1: To design and implement a simple class RandIndexQueue<T> that will act as a simple data structure for accessing Java Objects. Your RandIndexQueue<T> class will primarily implement 3 interfaces – MyQ<T>, Indexable<T> and Shufflable. The details of these interfaces are explained in the files MyQ.java, Indexable.java and Shufflable.java. **Read these files over very carefully before implementing your RandIndexQueue<T> class.** See also more details below.

Goal 2: To utilize your RandIndexQueue<T> class by implementing a very primitive 2-person version of the card game "Blackjack". In this case you will write a subclass of RandIndexQueue<T> called BlackjackCards and your program will utilize BlackjackCards objects for the deck and hands of cards in the game. See more details below.

Details 1:
In Lecture 4 we discussed the author's QueueInterface<T>, which is an ADT that allows for adding at the logical back (enqueue) and removing from the logical front (dequeue) of the data structure. The methods in this interface are specified in file QueueInterface.java. See the Lecture 4 Powerpoint presentation for some background and ideas about the QueueInterface<T>.

In Recitation Exercise 1 you implemented (or will implement) two simple classes called PrimQ1<T> and PrimQ2<T> that satisfy QueueInterface<T> but in an inefficient way. Specifically, they use an array that maintains one logical end or the other of the queue at index 0 and thus requires shifting for one of the enqueue() or dequeue() operations. We will discuss specific run-time analysis of these implementations a bit later in the course, but it is intuitive that there is a lot of overhead in both of these implementations.

A queue can be implemented in a more efficient way with an array if we allow both logical sides of the queue to move along the array in a circular fashion. For example, consider the array below:

0	1	2	3	4	5	6	7
		10	20	30	40	50	

front back

In this queue, both front and back will move forward within the array as we enqueue() or dequeue() in the queue. For example, if we enqueue(55) in this queue, it will look as follows:

0	1	2	3	4	5	6	7
		10	20	30	40	50	55

front back

If we then dequeue(), the queue will look as follows:

0	1	2	3	4	5	6	7
			20	30	40	50	55

front back

Note that for this approach to work effectively, both back and front will need to "wrap" around the end of the array when necessary. This enables the beginning indices in the array to be reused. For example, if we now enqueue(66), the array will appear as follows:

0	1	2	3	4	5	6	7
66			20	30	40	50	55

back front

Note that when the queue is implemented in this way, we do not have to shift data in the array and either of the enqueue() or dequeue() methods can be implemented with just a few statements. However, there are some special cases to consider (ex: detecting a full array, handing an empty queue) so think carefully about how you would implement your class.

In this assignment you will implement the class RandIndexQueue<T>, which is required to have the following functionality:

1) RandIndexQueue<T> will implement the MyQ<T> interface, which extends QueueInterface<T>. MyQ<T> adds some extra methods to QueueInterface<T>, which are explained in the file MyQ.java. Read over this file carefully to see the details of all of the additional methods. Also be sure to read QueueInterface.java to see the requirements of the original queue methods. Note also that you will need the QueueInterface.java file in order to compile and run your programs.
2) RandIndexQueue<T> will also implement the Indexable<T> interface. This interface has methods that allow indexing of the data structure – so that we can get and set values at specific locations. For details on Indexable<T>, see file Indexable.java. Read over this file very carefully to see the methods and their requirements.
3) RandIndexQueue<T> will also implement the Shufflable interface. This interface has one method – shuffle() – which will re-organize the data within the structure in a pseudo-random way. For details on Shufflable, see file Shufflable.java. Read over this file very carefully to see the requirements of the shuffle() method.

Thus, your RandIndexQueue<T> class should have the following header:

```
public class RandIndexQueue<T> implements MyQ<T>, Shufflable, Indexable<T>
```

The data in your RandIndexQueue<T> **must be a regular Java array** (**not an ArrayList<T>** or similar class). You will also need some integers to keep track of your data (ex: the front and back of the queue and the logical size of the queue).

The array in your RandIndexQueue<T> class should be initialized of a specific length in a RandIndexQueue<T> constructor that takes an int argument:

```
public RandIndexQueue(int sz)
```

This will create an array of length sz to be used for your RandIndexQueue<T>. If this array fills you should **dynamically resize it** by creating a new array of double the size, and copying the data into the new array. **Be very careful with your array resizing.** Because of the circular nature of your array access, you cannot just copy the data in the old array into the same locations in the new array. Think about why this is the case. The important issue is that the queue after you resize the array is **logically equivalent** to the queue prior to resizing. For general ideas about resizing see course lectures and also see Sec. 2.34-2.43 in the Carrano text. In order to test that you are resizing appropriately, the MyQ<T> interface includes the method capacity(), which will return the number of locations in the underlying data structure (in this case, the array).

To allow for testing and more functionality in your class, you must also implement the following 3 methods in your RandIndexQueue<T> class:

```
public RandIndexQueue(RandIndexQueue<T> old)
public boolean equals(RandIndexQueue<T> rhs)
public String toString()
```

The first method is a copy constructor, which should make a new RandIndexQueue<T> that is logically equivalent to the argument. The copy constructor should not be shallow – the arrays should not be shared. However, it does not have to be completely deep either (you don't need to make new copies of the individual items in the queue). Note also that the copy does not have to have the same values for front and back as the original – the important thing is that both contain the same data in the same relative ordering from front to back.

The second method is an equals method that returns true if the queues are logically equivalent. In this case logically equivalent means that the individual items in both queues are equal() and in the same relative positions within the queues (based on front and back). However, they do not have to be located in the same index values in the arrays. This method will assume that a reasonable equals() method exists for whatever type T is being used for the RandIndexQueue<T>.

The third method is a toString() method which will make a single String of all the data in the RandIndexDeque<T> from front to back and return it. This method will assume that a reasonable toString() method exists for whatever type T is being used for the RandIndexQueue<T>.

I recommend a LOT of pencil and paper work before actually starting to write your code.

After you have finished your coding of RandIndexQueue<T> class, the **Assig1A.java** file should compile and run correctly, and should give output identical to the output shown in file A1A-out.txt (except for the segments where the data is shuffled, since it will be pseudo-random in that case). See comments in Assig1A.java for some more help with developing your RandIndexQueue<T> class.

Details 2:
Blackjack is a card game that is popular in casinos and online. In the real game of Blackjack players risk money and the strategies and options for playing a hand are quite complex. However, in this assignment you are only required to implement a very primitive version of Blackjack with one player, a dealer, and much simplified rules. Here are the rules that you must follow:

- The ***shoe*** of cards to be used for the game will consist of some multiple of regulation 52-card decks (with the multiple to be determined when the program executes). For example, a game could have 1 deck in the shoe or 6 decks in the shoe. Initially the cards in the shoe must be shuffled.
- Each card in the shoe has a ***value***: numbered cards have their numeric values; Jacks, Queens and Kings have the value 10; Aces can have either the value 11 or the value 1. See the files Card.java and A1Help.java for more information on the cards and their values.

- Each game will consist of a number of ***rounds***. The number of rounds will be determined when the program executes.
- The player and the dealer will each have a ***hand*** for each round, which consists of cards dealt from the shoe.
- Each hand has a **value** which is the sum of values of the cards in the hand. Note that for a hand with one or more Aces, more than one value for the hand is possible.
- Both the player and dealer are initially dealt two cards (alternating) from the shoe.
- The goal for the value of a hand is 21. If either the dealer or player scores a 21 in the initial 2 cards, that hand is called a ***blackjack***. A blackjack beats all other hands, except another blackjack, which it ties.
- If neither the dealer nor player has a blackjack, play continues in the following way:
  - The player will ***hit*** (draw cards from the shoe) until the value of the player's hand is at least 17 (i.e. player will hit on a 16 but not on a 17 or greater). If a hit takes the player's value above 21, the player will ***bust*** and lose the round; otherwise the player will ***stand*** with their value.
  - If the player did not bust, the dealer will hit following the same rules as the player – taking hits until the value of the dealer's hand is at least 17. If a hit takes the dealer's value above 21, the dealer will bust and lose the round; otherwise the dealer will stand with their value.
  - For either the player or the dealer, initially an Ace is valued as 11, but if that value will cause them to bust, then the value of 1 will be used instead. For example, if the player is dealt [5, Ace, 3] then the Ace counts as 11 and the hand value is 19. However, if the player is dealt [8, 6, Ace], then the Ace will count as 1, and since the total value will then be 15, the player in this case would actually take another hit. Note that the order that the cards are received is important in this process. In the second example above, if the player had received [8, Ace] they would have stood at 19 without taking the extra hit, since the 11 value for an Ace is initially used.
  - If neither the player nor the dealer busts, then the hand with the highest value will win the round. If both hands have the same value then the round is a tie.
- At the end of each round, all cards that were drawn from the shoe are placed into a ***discard*** pile.
- At the end of each round, the size of the shoe is checked. If it is at ¼ its original size or less, the following actions are taken:
  - All of the cards in the discard pile are added back into the shoe
  - The cards in the shoe are shuffled

  In order to facilitate grading, you must output a statement when you reshuffle the cards in the shoe. See Blackjack-out.txt for an example showing this.
- Once all rounds in a game have been played, the following will be displayed:
  - Total rounds played
  - Number of rounds won by the dealer
  - Number of rounds won by the player
  - Number of ties

More Important Blackjack Details: The following implementation requirements for your Blackjack game **must be followed**:

- You must write a subclass of RandIndexQueue<T> called BlackjackCards and use that type for your deck, hands and discard pile throughout the game. In particular, you should have the following declaration:

  ```
  public class BlackjackCards extends RandIndexQueue<Card>
  ```

  Note that this class will inherit all of the public methods from RandIndexQueue<Card> and you must add one additional method to this class:

`public int getValue()`
This method will return a single value for the current BlackjackCards object, based on the evaluation discussed above, and the getValue() method will be utilized in order to determine the values of the player's and dealer's hands during the game. Think carefully how this method will be implemented.

- Your BlackjackCards class should run with main program Assig1B.java and the output must match that shown in file A1B-out.txt.
- As mentioned above, your shoe, discard pile, dealer hand and player hand must all be BlackjackCards objects and access to them must be done exclusively through the functionality of the BlackjackCards type. For example, you are **NOT** allowed to copy the Card objects into an array or some other collection of data in your program.
- The Card class is provided for you in file Card.java and must be utilized as is with no changes.
- The Card objects generated when you initialize your shoe must be the only Card objects used throughout your program. These cards can be moved between the various BlackjackCards objects, but **once the game has started no new Card objects can be created in your game**. Think of a real card game – the same cards are used over and over throughout the game.
- The number of rounds to be played and the number of decks used in the shoe must both be passed into the program as command line arguments.
- To help with the grading of your program, **a third command line argument must also be used.** This value will be the number of rounds in the execution that will show "trace" output. In the trace output execution of your program, the details of each round must be shown to the user – including the contents and values of each hand and who wins the round. For example, here are some command line executions and what they mean:
  `> java Blackjack 200 4 10` → 200 rounds played with 4 shoes, tracing the first 10
  `> java Blackjack 100 6 20` → 100 rounds played with 6 shoes, tracing the first 20
- For details on how your trace version should look, see Blackjack-out.txt.

You must submit **in a single .zip file** containing (minimally) the following 11 complete, working source files for full credit:

MyQ.java QueueInterface.java Indexable.java Shufflable.java
Assig1A.java Assig1B.java EmptyQueueException.java Card.java

the **above eight files** are given to you and must not be altered in any way.

RandIndexQueue.java BlackjackCards.java Blackjack.java

the **above three files** must be created so that they work as described. If you create any additional source or text files, be sure to include those as well. **Do not submit any project or .class files.**

You must also submit an Assignment Information Sheet for this (and every) assignment. In this file you will list some information that will aid your TA in grading your assignment. For a sample Assignment Information Sheet, see infosheet.html. You may create your own information sheet if you wish, but it should contain the same information contained in infosheet.html.

The idea from your submission is that your TA can unzip your .zip file, then compile and run all of the main programs (Assig1A.java, Assig1B.java and Blackjack.java) **from the command line** WITHOUT ANY additional files or changes, so be sure to test it thoroughly before submitting it. If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why (ex: "I could not get the shuffle() method to work, so I eliminated code that used it") on your Assignment Information Sheet. You will lose some credit for not getting it to work properly, but getting the main programs to work with modifications is better than not getting them to work at all. **Note: If you use an IDE such as Eclipse to develop your programs, make sure they will compile and run on the command line before submitting – this may require some modifications to your program (such as removing some package information).**

Hints / Notes:

- See program A1Help.java for some help with the Card class and using the RandIndexQueue<T> class with Card objects.
- See file A1A-out.txt to see how your output for Assig1A should look. As noted, your output when running Assig1A.java should be identical to this with the exception of the order of the values after being shuffled.
- See file A1B-out.txt to see how your output for Assig1B should look. As noted, your output when running Assig1B.java should be identical to this.
- See file Blackjack-out.txt for some example runs of my Blackjack.java program. Your Blackjack program output does not have to look exactly like this but the functionality should be the same. Note in particular the information shown during the trace output and the notices of the shuffling of the shoe.
- At any time your RandIndexQueue<T> class will have a logical size (# of items being stored) and a physical size (length of the array). Be careful to distinguish between these sizes. For example, operations such as toString(), get() and set() will be utilizing the logical size rather than the physical size.

Extra Credit:

- Clearly the Blackjack game here is very primitive and leaves out most of the strategy involved in real Blackjack. You can get some extra credit if you improve the logic to make the player more competitive. This is non-trivial so don't feel obliged to completely implement a "good" player strategy. To see strategies Google "Blackjack strategy".
- This version of the program is a batch program with no player interaction. Add an interactive version of the game so the player can bet and play their own favorite strategy.
- You can earn **at most 10 points of extra credit** no matter what you do. Either of the options above will likely have much more effort than reward. Keep this in mind before trying any extra credit!