COMPUTER SCIENCE 21A (SPRING TERM, 2017)
DATA STRUCTURES

PROGRAMMING ASSIGNMENT 1

Implementing Uno with Linked Lists

due Wed, March 1 @ 11:30pm

This assignment will test your ability to implement many versions of the LinkedList and other data structures, as well as use your implementation in practice.

Rules

If you haven't played the game of Uno before, here is a quick rundown of the rules (slightly modified).

First, every player sits in a circle (which you will code as a circularly doubly-linked list).
You have a deck of Uno cards (UnoDeck.java), and each player draws 7 cards into their hand.
After that, one card is taken from the deck, and placed in the discard pile.

The cards are of many different types: There are some with a color and a number, there are some with a color and an action, there are wild cards that can be used as any color. Each player takes a turn, and places a card from their deck into the discard pile. The card that the player puts down has to have one of the following properties:

1. It has to be the same color as the last card placed down.
2. It has to be the same number as the last card placed down.
3. It has to be the same type of special card as the last card placed down (e.g. they're both reverse cards, both draw two cards, etc…).
4. It is a wild card, or the previous card was a wild card (note: this is a simplification of regular uno rules).

If a player has no cards that are able to be placed down, they have to draw a card, and their turn is skipped.

When a special card is placed down, there are some special things that happen:

- If a reverse card is placed down, the order of the player's turns is reversed.
- If a draw two card is placed down, the next player has to draw two cards.
- If a skip card is placed down, the next player gets skipped.
- If a wild draw four is placed down, the next player has to draw four cards.

A player wins when they have no more cards in their hand.

Implementation

UnoGame.java - Main Method

Implement the Uno game in the following way:

1. In the beginning, have a prompt that asks the user to enter the names of the people that are playing. For the first five names, add the player to the PlayerCircle in alphabetical order. Then after that, add the rest of the players to a Queue (you can ask the user how many players they're expecting).
2. For each player, draw 7 cards and add them to the player's hand. Print out each player's hand.
3. Draw one card from the deck, and add it to the discard pile. Say what card was placed down.
4. Then, for each round (starting with the first player):
    a. If the previous card was a draw 2 or draw 4, draw those cards.
    b. Tell the user which of their cards they can place down. If they have none, draw a card, and skip to the next player. Use the canBePlacedOn method to make this much easier.
    c. Allow the user to select a card from one of the ones that can be legally placed.
    d. Take that card from the user's hand, and discard it into the deck.
    e. If the player has no more cards, they won.
    f. Go to the next player (if the card placed was a reverse card, reverse the direction of which way you are going. If it's a skip card, skip the next player).
5. Once the round is over, report the winner, and how many times around the circle you went. **Because of reverse cards, you can choose how to calculate this metric, but comment with your reasoning**. Also, report the loser (the one with the most cards), and take them out of the circle, putting them in the queue. Take someone else out of the queue, put them in the PlayerCircle, and then start again.

Classes to Create

The following class has already been created for you, and you don't need to touch.

UnoCard.java

Has the following methods which are important to you:

- String getColor() - returns color of card as string (red, green, yellow, blue, wild)
- int getNumber() - returns number of card, -1 for special cards
- boolean isSpecial() - returns true if the card is a special card
- boolean isDrawTwo() - returns true if the card is a draw two card
- boolean isReverse() - returns true if the card is a reverse card
- boolean isSkip() - returns true if the card is a skip card
- boolean isWild() - returns true if the card is a wild card. This is also true for wild draw four cards.
- boolean isWildDrawFour() - returns true if the card is a wild draw four card. Note that if this is true, then isWild is also true.

- boolean canBePlacedOn(UnoCard other) - returns true if your card can be placed on top of “other” in the discard pile. This makes things MUCH easier :)
- String toString() - for debugging
- boolean equals(UnoCard other) - says if other is the same card as you.

You need to implement the following classes:

SinglyLinkedNode<T>

Nodes for the SinglyLinkedList, described below. We will make it generic, though it will only be used for Uno cards in this PA.

1. SinglyLinkedNode(T data) - constructor that initializes the node.
2. T getData() - returns the data in the node
3. void setNext(SinglyLinkedNode<T> nextNode) - sets the node as the “next” node in the list, returned by getNext()
4. SinglyLinkedNode<T> getNext() - returns the next node in the list
5. toString() - return the string of the data inside it.

SinglyLinkedList<T>

This will be how we will represent each player’s hand, as well as the uno deck and discard pile.

1. SinglyLinkedList() - constructor (optional)
2. SinglyLinkedNode<T> getHead() - returns the first node in the list (null if empty)
3. void regularInsert(T data) - insert “data” at the end.
4. void randomInsert(T data) - insert “data” at a random point in the LinkedList. It should be equally likely that a card will end up at any of the possible locations, including the front and the end. This is going to be used when inserting a **card into the discard pile, as well as initially into the deck.**
5. T remove(T data) - delete the “data” node from the list.
6. **T removeIndex(int index) – delete the item at “index”, return that item.**
7. int size() - gets size of linked list.
8. toString()

UnoDeck

This class encompasses the abstraction of drawing and discarding cards. Part of it has already been started for you. Make sure to instantiate the deck and discard linked lists at the top.

- public unoDeck() - constructor which creates an uno deck with all the uno cards. The scaffolding has already been created for you, and lots of new cards have been created, in the right way. Your job is just to add the new Card’s to the deck. Make sure you add them with the randomInsert method, so that the deck is shuffled when it’s first used.
- UnoCard getLastDiscarded() - gets the last card which was discarded - use this to compare to the card you’re about to put down.
- UnoCard drawCard() - draw a card from the deck. If there are no more cards in the deck, it moves all of the cards from the discard pile to the deck (the discard pile will already be shuffled, as described below).
- void discardCard(UnoCard c) - adds c to the discard pile, and sets it as the last discarded card. Will throw an error if an invalid card is placed on the deck. **You can choose how to throw the error, but make sure in the client code that no invalid cards will be**

placed. When calling this, **use the randomInsert method, so that the discard pile will be randomly shuffled.**

- String toString()

Player

This method has been partially written for you. Notice the getNextPlayer, getPrevPlayer, setPrevPlayer, and setNextPlayer methods (which you'll need for the PlayerCircle and for the game). However, you have to add the following things:

1. Add a field for a singly linked list at the top (for the hand), and instantiate it in the constructor. This will be for the player's hand.
2. Implement addToHand(UnoCard c) - adds c to the player's hand.
3. Implement removeFromHand(int index) - removes the item at the passed in index from the hand. **Note that the method signature has changed, this should return the item at that index that was removed.**
4. Implement winner()
5. Feel free to change the toString method to something better.

PlayerCircle

This will be a circle of players. If you didn't notice, each player has a getNextPlayer, getPrevPlayer, setNextPlayer, and setPrevPlayer. With this class, you need to to design a way to add the players, such that all players will be in a "circle." E.g., if there are three players, player 1 is next to player 2, who is next to player 3, who is next for player 1. This is basically a doubly-circularly linked list.

1. Constructor (optional)
2. addToCircle(Player p) - add the player p to the circle, such that the players are still sitting in a circle. Add them in alphabetical order **(you can use the string compareTo method for this)**.
3. Player getFirstPlayer() - returns the first person in the circle.
4. int getSize() - returns number of players in the circle.
5. toString() - print all the players in the circle

Queue<T>

This will be an array implementation queue for the extra players. **See this Latte post about how to see how to use generics in an Array.** You need to implement the following methods:

1. queue(int size) - Constructor that creates the internal array of size "size", as well as any other variables needed in the queue.
2. void enqueue(T data) - enqueue data
3. T dequeue() - dequeue first item in the queue
4. int getSize() - return size of the queue
5. boolean isEmpty() - returns true if queue is empty
6. boolean isFull() - return true if queue is full.

Submission:

Submit via Latte a zip file with all the following files:

- UnoGame.java
- UnoCard.java
- SinglyLinkedNode.java
- SinglyLinkedList.java
- UnoDeck.java
- Player.java
- PlayerCircle.java
- Queue.java

The zip file should be titled <your-username>-PA1.zip - for example, if your brandeis email is dilant@brandeis.edu, your zip file should be called dilant-PA1.zip

IMPORTANT

1. Your code should be well commented:
   - Add your name and email address at the beginning of each .java file.
   - Write comments within your code when needed.
     - You must use Javadoc comments for your classes and methods.
     - You must write each method's running time in your Javadoc comments for that method.
2. Excluding arrays, you may only use data structures that you implement yourself.
   - Your data structures should not have hardcoded data types. For example, your implementation of a stack should be able to handle Integers, Doubles, Strings, or any sort of Java object.