READ ME

1. Container
   1. LLStack
      1. This makes a generic Stack implemented as a Linked List
      2. It implements StackSpecs which is an interface containing all the methods needed for a stack.
      3. top - null, until another node is pushed.
      4. stackSize – variable that increments when pushing, and decrements when popping.
      5. isEmpty – if top = null, then its empty
      6. emptyStack – pops all values until isEmpty = true
      7. push – sets a new Node as the top with its data being the input parameter, and nextNode being the previous top. Also increments stackSize.
         1. If input parameter, throws nullPush exception, that doesn’t make a new node, and prints error message.
      8. pop – makes a variable popData to hold top’s data. Sets top’s next node (One underneath it) to top, and decrements stackSize.
         1. If isEmpty = true, throws emptyStack exception that doesn’t pop anything and prints error message.
      9. peek – returns the tops.getData
         1. If isEmpty = true, throws emptyStack exception that doesn’t return anything and prints error message.
      10. getStackSize – returns stackSize
      11. toString – creates a string that starts with “[ ” and concats all the String values of the popped Node’s data with a newline between each popped Node and ends with a “ ]”. But because we are popping values, we place the popped Nodes into a temporary Stack and then we import them all back into the original stack when done.
   2. Node
      1. data – this is a generic type that can hold objects that contain data
      2. nextNode – is a node in itself that is linked to the original node.
      3. When creating a node, and not given a nextNode, we set nextNode to null.
      4. Otherwise, when given everything, we set the data and nextNode.
      5. We also have getters and setters for data and nextNode
   3. StackQ (will talk about enqueue and dequeue first because it will make everything else make sense)
      1. StackQ holds two stacks { enqueueStack, dequeueStack }.
      2. The constructor calls the contructor of LLStack on enqueueStack and dequeueStack.
      3. Enqueue – pops all values that are in dequeueStack and pushes them into the enqueueStack. Then pushes the enqueueObj given as an input parameter into enqueueStack. Then it pops all its nodes and pushes them back into the dequeueStack. This will have the last node in to be on the top of the enqueueStack but then on the bottom of the dequeueStack to have it pop last.
         1. Has the same exception as the LLStack’s push method.
      4. Dequeue – pops off top node of dequeueStack which was the first element to enqueue.
         1. Has the same exception as the LLStack’s pop method.
      5. Peek – peeks at the top node of the dequeueStack.
         1. Has the same exception as the LLStack’s peek method.
      6. isEmpty - Because at all times, dequeueStack is filled and enqueueStack is only used to reverse and enqueue, but then reversed right back into dequeueStack, isEmpty looks to whether or not the dequeueStack isEmpty is true.
      7. emptyQueue – this will call the emptyStack on the dequeueStack.
      8. toString – calls the toString on the dequeueStack.
      9. getStackSize – return the stackSize of dequeueStack
      10. getters for the Stacks returns the two stacks.
2. Data
   1. DataClass
      1. This holds a dataName and dataInt.
      2. It has getters and setters to access the private variables
      3. toString – creates a string that will concat as “Data Name: [dataName] Data Int: [dataInt]”.
         1. I ran a loop so that the spacing in between the D in Data Name and D in Data int is exactly 25 spaces exactly. So when printing the Linked List, it will format like below:
            1. Data Name: Example Data Int: 1
            2. Data Name: Ex Data Int: 2
3. Interfaces
   1. QueueSpecs
      1. Holds the Methods { isEmpty, emptyQueue, enqueue, dequeue, peek} because all Queue’s should have this functionality.
   2. StackSpecs
      1. Holds the Methods { isEmpty, emptyStack, push, pop, peek} because all Stack’s should have this functionality.
4. Driver
   1. Driver
      1. Input – scanner object that takes in input
      2. Queue – StackQ object that will be tested on
      3. DisplayMenu – displays the menu, then asks for selection and calls the choice method
      4. Choice – choice = Scanner Input
      5. If choice = {a,b,c,d,e,f,g} it does its respective function.
         1. A) enqueue – asks for dataName and dataInt to input into a DataClass object. Then we enqueue that DataClass Object into queue.
            1. If dataInt’s input is not an int, it will give an exception, that will be defined at the end of choice(). It’s error message says that the input is unrecognized and to try again.
         2. B) dequeue – displays the returned value of queue.dequeue()
         3. C) peek - displays the returned value of queue.peek()
         4. D) Display Queue – Will print queue so it will look like:
            1. [ Data Name: Example Data Int: 1  
                Data Name: Ex Data Int: 2 ]
         5. E ) Will display the two stacks of the queue. The enqueueStack will always be empty because it is only used as a temporary holder for dequeueStack.
         6. F) This will call queue.getStackSize ()
         7. G) This will exit
         8. If input is not either of these it throws an exception with the error Message saying that the input is unrecognized and to try again.
         9. If an Exception is thrown, the method displays the message. Clears out the input Scanner by redeclaring it. Then takes you back to the displayMenu.

Enqueue(“1”):



ENQ

5

6

7

8

1

DEQ

DEQ

1

8

7

6

5

ENQ

DEQ

8

7

6

5

ENQ

5

6

7

8

DEQ

ENQ

🡪 🡪 🡪

Now the Enqueued “1” is going to the last to pop(), like a queue.

Dequeue() :

6

7

8

1

DEQ

ENQ

ENQ

5

6

7

8

1

DEQ

🡪 Returns “5”