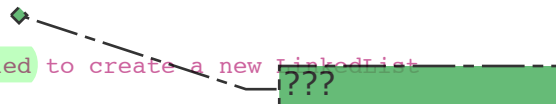
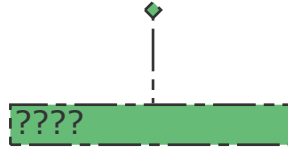


```

1  /**
2  Code for the creation and manipulation of a Data Structure called a LinkedList which is essentially a
3  @author Yanni Angelides
4  @version 11/30/15
5  */
6
7  import java.util.Iterator;
8  import java.lang.Iterable;
9  import java.util.NoSuchElementException;
10
11 public class LinkedList<E> implements Iterable<E> //Stack<E>, Queue<E>
12 {
13     private ListNode head;
14     private ListNode tail;
15     private int size;
16     //singley link
17     //head pointer
18     //tail pointer
19
20     /**
21     Creates a completely empty LinkedList
22     */
23     public LinkedList()
24     {
25         head = null;
26         tail = null;
27         size = 0;
28     }
29
30     /**
31     Creates a LinkedList that contains one ListNode
32     @param the ListNode that is used to create the LinkedList
33     */
34     //Precondition that h.getNext() == null
35     public LinkedList(ListNode<E> h)
36     {
37         head = h;
38         tail = h;
39         size = 1;
40     }
41
42     /**
43     Creates a LinkedList that is
44     @param LinkedList that is copied to create a new LinkedList
45     */
46     public LinkedList(LinkedList<E> list)
47     {
48         if (list == null)
49         {
50             throw new IllegalArgumentException(); //cannot copy a LinkedList that is null
51         }
52         else
53         {
54             for (ListNode<E> i = list.get(0); i != list.get(list.size()); i = i.getNext()) //starts at
55             {
56                 if(size == 0) //if size == 0 then you have to set the head to whatever ListNode you are
57                 {
58                     i.setNext((list.get(0)).getNext());
59                     head = i;
60                     size++;
61                 }
62                 else
63                 {
64                     i.setNext(i.getNext()); //sets the next of the ListNode being created to the next
65                     size++;
66                 }
67             }
68             tail = list.get(list.size());
69             size++;
70         }
71     }
72
73     /**
74     A helper method that can get the ListNode at a specific index in the LinkedList

```



Isn't this method supposed to be copying the contents of list?

```

75 @param int index of the ListNode that is needed
76 @return ListNode the ListNode that is needed
77 */
78 public ListNode<E> get(int indx)
79 {
80     if (index >= size || index < 0)
81     {
82         throw new IndexOutOfBoundsException();
83     }
84     else
85     {
86         ListNode trav = head;
87         for (int i = 0; i < indx; i++) //The for loop will start at the head of the LinkedList and
88         {
89             trav = trav.getNext();
90         }
91         return trav;
92     }
93 }
94
95 /**
96 Finds the index of a specified ListNode within the LinkedList
97 @param ListNode<E> that needs to be found
98 @return int index of the ListNode<E> passed in as a parameter
99 */
100 public int indexOf(ListNode<E> n)
101 {
102     ListNode trav = head;
103     for (int i = 0; i < size; i++)
104     {
105         if(trav == n) //same for loop as the one in the above method except that each time it goes
106         {
107             return i;
108         }
109         trav = trav.getNext();
110     }
111     return 0;
112 }
113
114 /**
115 @return int size of the LinkedList
116 */
117 public int size()
118 {
119     return size;
120 }
121
122 /**
123 Completely empties the LinkedList so that there are no ListNodes in it
124 */
125 public void clear()
126 {
127     head = null;
128     tail = null;
129     size = 0;
130 }
131
132 /**
133 Sets a specific index of the LinkedList to an E passed in as a paramater
134 @param int index that needs to be changed, E item that you want to set the specified index to
135 */
136 public void set(int indx, E item)
137 {
138     if (index >= size || index < 0)
139     {
140         throw new IndexOutOfBoundsException();
141     }
142     }
143     ListNode<E> n = new ListNode<E>(item); //Need to make the item into a ListNode so that it can
144     if(indx == 0)
145     {
146         n.setNext(head.getNext()); //if indx == 0 then the item parameter must become the head
147         head = n;
148

```

```

149     }
150     else if(indx == size)
151     {
152         get(size - 1).setNext(n); //if indx == size the item parameter must become the new tail
153         tail = n;
154     }
155     else
156     {
157         (get(indx - 1)).setNext(n);
158         n.setNext(get(indx+1));
159     }
160 }
161
162 /**
163  Checks if the item passed in as a parameter is contained in the LinkedList
164  @param E item that the LinkedList needs to be checked for
165  @return boolean indicating if the object is contained within the LinkedList
166  */
167 public boolean contains(E item)
168 {
169     for(ListNode<E> i = head; i != tail; i = i.getNext())
170     {
171         if(i.getValue() == item)
172         {
173             return true;
174         }
175     }
176     return false;
177 }
178
179 /**
180  Takes out a specific item from the LinkedList
181  @param E item that needs to be removed
182  @return boolean indicating if the item was removed or not
183  */
184 public boolean remove(E item)
185 {
186     if(head.getValue() == item)
187     {
188         removeFirst();
189     }
190     else if(tail.getValue() == item)
191     {
192         removeLast();
193     }
194     else
195     {
196         for(ListNode<E> i = head; i != tail; i = i.getNext())
197         {
198             if((i.getNext()).getValue() == item) //finds the item within the LinkedList
199             {
200                 i.setNext((i.getNext()).getNext()); //sets the next class field of the ListNode to be
201                 size--;
202                 return true;
203             }
204         }
205         return false;
206     }
207     return false;
208 }
209
210 /**
211  Removes the ListNode at a certain index in the LinkedList
212  @param int index that needs to be removed
213  @return boolean indicating whether or not the index was removed
214  */
215 public boolean remove(int indx)
216 {
217     if (indx >= size || indx < 0)
218     {
219         throw new IndexOutOfBoundsException();
220     }
221     if (indx == 0)
222     {

```

This won't work.

Need to use .equals

Remember, == compares memory address. The .equals method actually compares the values.

```

223         this.removeFirst();
224         return true;
225     }
226     if (indx == size)
227     {
228         this.removeLast();
229         return true;
230     }
231     else
232     {
233         (get(indx - 1)).setNext((get(indx)).getNext());
234         size--;
235         return true;
236     }
237 }
238
239 /**
240 Adds a specific item to the LinkedList at a specified index
241 @param int indx, index where the item needs to be added, E item that needs to be added
242 */
243 public void add(int indx, E item)
244 {
245     if (index > size || indx < 0)
246     {
247         throw new IndexOutOfBoundsException();
248     }
249     if (indx == 0)
250     {
251         this.addFirst(item);
252     }
253     if (indx == size)
254     {
255         this.addLast(item);
256     }
257     else
258     {
259         ListNode<E> n = new ListNode<E>(item);
260         ListNode<E> node = head;
261         for(int i = 0; i < indx - 1; i++)
262         {
263             node = node.getNext();
264         }
265         n.setNext(node.getNext());
266         node.setNext(n);
267     }
268     size++;
269 }
270
271 /**
272 Adds the specified item to the end of the LinkedList
273 @param E item that needs to be added
274 */
275 public void add(E item)
276 {
277     if (head == null) //Just in case the LinkedList is empty
278     {
279         this.addFirst(n);
280     }
281     else
282     {
283         this.addLast(n);
284     }
285     size++;
286 }
287
288 /**
289 Creates a String representation of the LinkedList
290 @return String that represents the LinkedList
291 */
292 public String toString()
293 {
294     String str = " ";
295     for (ListNode<E> i = head; i != tail; i = i.getNext())
296     {

```

There is no n in this method. Do you mean item?

```

297         str += i.toString(); //this toString method is for ListNode because i is a ListNode, met
298     }
299     return str;
300 }
301
302 /**
303 Removes the first ListNode in the LinkedList
304 @return ListNode removed from the LinkedList
305 */
306 public ListNode<E> removeFirst()
307 {
308     ListNode<E> save = head; //have to save head here because once it is removed from the LinkedL
309     head = head.getNext();
310     size--;
311     return head;
312 }
313
314 /**
315 Removes the last ListNode from the LinkedList
316 @return ListNode that was removed
317 */
318 public ListNode<E> removeLast()
319 {
320     ListNode<E> trav = head; //have to traverse to the back of the LinkedList because you can't j
321     for(int i = 0; i < size - 1; i++)
322     {
323         trav = trav.getNext();
324     }
325     ListNode<E> save = tail;
326     tail = trav;
327     size--;
328     return tail;
329 }
330
331 /**
332 Adds an item to the beginning of the LinkedList
333 @param E item that needs to be added
334 */
335 public void addFirst(E item)
336 {
337     ListNode n = new ListNode(item); //item must be turned into a ListNode before being added to
338     n.setNext(head);
339     head = n;
340     size++;
341 }
342
343 /**
344 Adds a specified item to the beginning of the LinkedList
345 @param E item that needs to be added
346 */
347 public void addLast(E item)
348 {
349     ListNode n = new ListNode(item);
350     tail.setNext(n);
351     tail = n;
352     size++;
353 }
354
355 /**
356 Checks if the LinkedList is empty (does not have any ListNodes in it)
357 @return boolean indicating whether or not the LinkedList is empty
358 */
359 public boolean isEmpty()
360 {
361     if(size == 0) //size will always be zero if the LinkedList is empty
362     {
363         return true;
364     }
365     else
366     {
367         return false;
368     }
369 }
370

```

```

371  /**
372  Method that allows for the creation of an Iterator of LinkedList so for loops can be used with it
373  @return Iterator<E> created in the LinkedListIterator class
374  */
375  public Iterator<E> iterator()
376  {
377      return new LinkedListIterator(head); //parameter is head because that is where the LinkedList
378  }
379
380  public static void main(String [] args)
381  {
382      LinkedList arr = new LinkedList();
383      ListNode<Integer> n1 = new ListNode<Integer>(5);
384      ListNode<Integer> n2 = new ListNode<Integer>(9);
385      ListNode<Integer> n3 = new ListNode<Integer>(234);
386      arr.add(n1);
387      arr.add(n2);
388      arr.add(1, n3);
389      System.out.println(arr.toString());
390  }
391  //add
392  //remove
393
394  //STACK
395
396  /**
397  Adds an item to the top of the stack
398  @param E item to be added to the stack
399  */
400  public void push(E item)
401  {
402      this.addFirst();
403  }
404
405  /**
406  Gets whatever ListNode is at the top of the stack
407  @return ListNode<E> at the top of the stack
408  */
409  public ListNode<E> peek()
410  {
411      return head;
412  }
413
414  /**
415  Removes the item at the top of the Stack
416  @return E item removed from the Stack
417  */
418  public E pop()
419  {
420      ListNode<E> n = new ListNode<E>();
421      n = this.removeFirst();
422      this.removeFirst();
423      return n.getValue();
424  }
425
426  //QUEUE
427
428  /**
429  Adds item to the end of the Queue
430  @param E item to be added to the Queue
431  */
432  public void offer(E item)
433  {
434      this.addLast();
435  }
436
437  /**
438  gets the last ListNode in the Queue
439  @return ListNode<E> that is at the end of the Queue
440  */
441  public ListNode<E> peek()
442  {
443      return tail ;
444  }

```

Did this actually run when you tested?

You're missing a semi-colon. Also, why do you need a ListNode?

```

445
446 /**
447 Removes the last ListNode in the Queue
448 @return E value of the last ListNode in the Queue
449 */
450 public E poll()
451 {
452     ListNode<E> n = new ListNode<E>()
453     n = this.removeLast();
454     this.removeLast();
455     return n.getValue();
456 }
457
458 /**
459 removeFirst()
460 E item = list.get(0)
461 list.add(0, item)
462 if(list.removeFirst() == item)
463
464 LinkedListIterator(ListNode<E> head)
465     curr = head
466
467 Declare stack and queue as a Linked List and test pull, pop, peek...
468
469 */
470 }
471
472 /**
473 This is the class that allows for the creation of an object that will act as the base of
474 the LinkedList
475 @author Yanni Angelides
476 @version 11/30/15
477 */
478
479 public class ListNode<E>
480 {
481     private E item;
482     private ListNode<E> next;
483
484     /**
485     Constructs a ListNode<E> with a specified item
486     */
487     public ListNode(E e)
488     {
489         item = e;
490         next = null;
491     }
492
493     /**
494     Constructs a ListNode with a specified item and pointer that points to the next object in the Lin
495     */
496     public ListNode(E e, ListNode<E> ln)
497     {
498         item = e;
499         next = ln;
500     }
501
502     /**
503
504     @return E value of the item in the ListNode
505     */
506     public E getValue()
507     {
508         return item;
509     }
510
511     /**
512     @return the next ListNode in the LinkedList
513     */
514     public ListNode<E> getNext()
515     {
516         return next;
517     }
518

```

```

519     /**
520
521     @param E the value that the item class field in the ListNode needs to be set to
522     */
523     public void setValue(E e)
524     {
525         item = e;
526     }
527
528     /**
529
530     @param the ListNode that the current ListNode is set to point to
531     */
532     public void setNext(ListNode<E> ln)
533     {
534         next = ln;
535     }
536
537     /**
538     @return String representation of the ListNode
539     */
540     public String toString()
541     {
542         String node = " ";
543         node += "Item: " + item.toString();
544         return node;
545     }
546 }
547
548 /**
549 Interface of the Queue class which is an offshoot of a LinkedList
550 @author Yanni Angelides
551 @version 11/30/15
552 */
553 public interface Queue<E>
554 {
555     void offer(E item);
556     // offer adds to the end of the Linked list so that the end pointer is always the same thing
557     E poll();
558     E peek();
559     boolean isEmpty();
560 }
561
562 /**
563 Interface of the Queue class which is an offshoot of a LinkedList
564 @author Yanni Angelides
565 @version 11/30/15
566 */
567 public interface Stack<E>
568 {
569     void push(E item);
570     // adds to the beginning of the list so that the end pointer is always the same thing
571     E pop();
572     E peek();
573     boolean isEmpty();
574 }
575
576 /**
577 LinkedListIterator class which is called by the iterator method in LinkedList so that for loops can b
578 @autor Yanni Angelides
579 @version 11/30/15
580 */
581 import java.util.NoSuchElementException;
582 import java.util.Iterator;
583 import java.lang.Iterable;
584
585 public class LinkedListIterator<E> implements Iterator<E>
586 {
587     private ListNode<E> curr;
588
589     public LinkedListIterator(ListNode<E> head)
590     {
591         curr = head;
592     }

```



```

593
594 /**
595 This is a method from the Iterator Interface that returns the next object in the Vector
596 @return E the next object of type E in the Vector
597 */
598 public E next()
599 {
600     if(hasNext() == false)
601     {
602         throw new NoSuchElementException(); //Because if hasNext() == false then there us no such
603     }
604     else
605     {
606         E item = curr.getItem();
607         curr = curr.getNext();
608         return item;
609     }
610 }
611
612 /**
613 @return boolean determining whether or not there is another object in the Vector
614 */
615 public boolean hasNext()
616 {
617     return curr != null;
618 }
619 }
620

```

I'd like you to update this code. I tried fixing your various compile errors, but they were too many. How did you actually test this code? Of particular concern is when you use ListNode instead of E and == instead of .equals. I'll commit the .java files I created out of your readme file. I fixed a few compile errors, but you should do the rest.