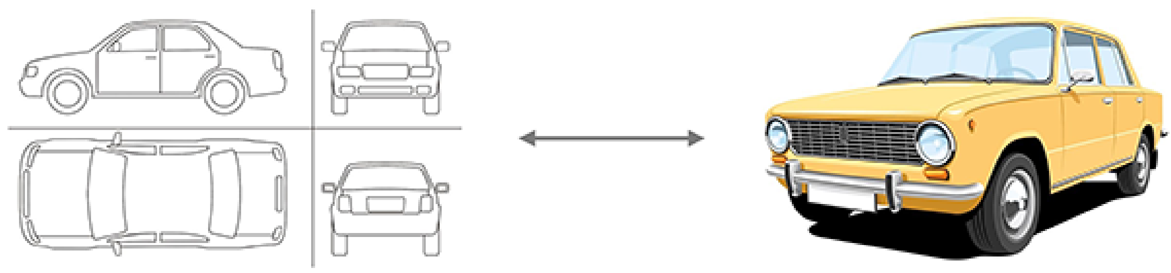
LinkedList vs ArrayList

The Java Collections Framework has two general-purpose classes for representing lists of things, namely `LinkedList` and `ArrayList`. In a previous article, we covered the `LinkedList` data structure. As the name implies, `LinkedList` gets its name from being internally based off a doubly-linked list.

Linked lists and java.util.LinkedList

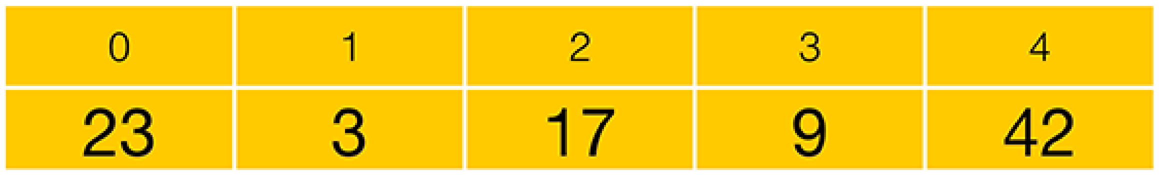
Linked lists and `java.util.LinkedLists` actually represent two different concepts. As an analogy, think of the difference between the abstract concept of a car on paper and a real BMW.

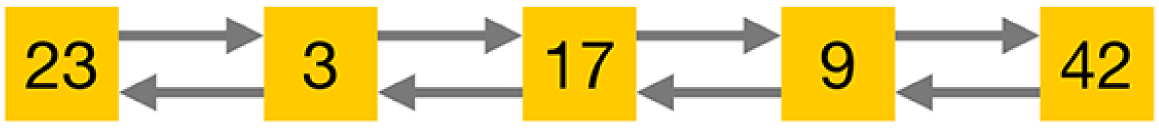


A linked list is an abstract concept of a data structure, independent of any programming language or platform, whereas the `LinkedList` Java class is a concrete implementation. Among other interfaces, `LinkedList` implements `java.util.List`. You can have duplicates in a `List` and you can go through each element in the same order as inserted.

Differences between `ArrayList` and `LinkedList`

As was covered in [a previous article](), both `ArrayList` and `LinkedList` implement the `java.util.List` interface, making them somewhat similar to each other, but that is where the similarity ends. First of all, `ArrayList` is based on raw Java arrays, while `LinkedList` is based on a doubly-linked list.





In contrast to an ArrayList, LinkedList's doubly-linked list allows more efficient insertions and removals of elements at any position within the list. Therefore, we prefer LinkedList over ArrayList whenever our main use of the list is to add and remove elements in random positions. Otherwise, ArrayList might be a better choice as storing elements in an array consumes less memory and generally offers faster access times. Besides implementing List, LinkedList also implements the Queue and Deque interfaces, giving it some additional functionality over ArrayList.

In conclusion, there is no overall winner between ArrayList and LinkedList. Your specific requirements will determine which class to use.

LinkedList Implementation

```
package java.util;

public class LinkedList implements List,Deque {
  private Node first;
  private Node last;

  public E get(int index) {…}
  public boolean add(E e) {…}
  public E remove(int index) {…}

  […]
}
```

Figure 1

Figure 1 shows a simplified code excerpt from the java.util.LinkedList class. A full grasp of every detail of the code excerpt is not needed. All that there is to it is to show that LinkedList is a normal Java class which anyone could have written, given enough time and knowledge. The full, actual source code is available online. After reading this article, having a look at it yourself is recommended. As is shown, LinkedList implements the List, Queue and Deque interfaces, as Deque extends Queue.

Next, we can see that LinkedList class has a reference to the first and the last elements of the list. Finally, we can also see that the class has functions such as *get*, *add* or *remove* – to access, insert or delete elements from the list.

As we just have just seen in the code, LinkedList's references to its first and its last elements are also shown as red arrows in Figure 2:

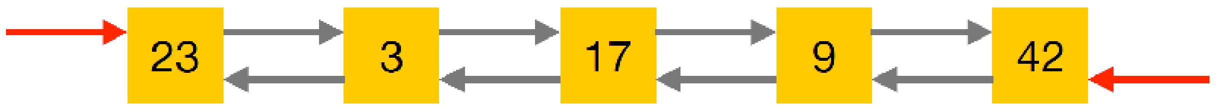


Figure 2

Every single element in a doubly-linked list has a reference to its previous and next elements as well as a reference to an item, simplified as a number within a yellow box in Figure 2.

```
public class Node {
  private E item;
  private Node previous;
  private Node next;

  public Node(E element, Node previous, Node next) {
    this.item = element;
    this.next = next;
    this.previous = previous;
  }

  […]
}
```

Figure 3

A code excerpt of a linked list node implementation is shown on Figure 3. It has private members for the item it holds, as well as for the previous and the next nodes in the list. As users of the Collections class `LinkedList`, we never directly access the nodes. Instead, we use the public methods `LinkedList` exposes which internally operate on the private Node members.

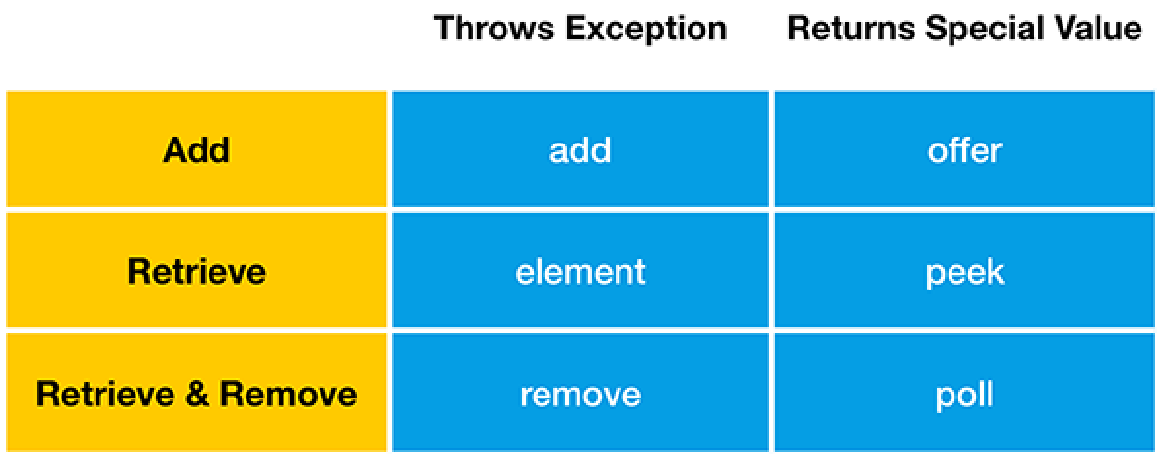
Having already covered the methods of the `List` interface in a previous article, we move on now to the methods of the Queue interface as implemented by `LinkedList`.

Queue

From a high-level perspective, the Queue interface consists of three simple operations: **add an element** to the end of the Queue, **retrieve an element** from the front of the Queue without removing it, and **retrieve and remove an element** from the front of the Queue.

In the lifetime of a queue, there are special situations like trying to remove an element from an empty queue or trying to add an element to a full, limited-capacity queue.

Depending on your specific implementation, this might be a fairly common situation. In this case, you will need a method that returns null or false. Alternatively, this might be an exceptional situation which requires you to have a method that throws an **Exception**. To this end, the Queue interface offers each of its operations in two flavors – one method that throws an Exception and another that returns a special value in certain cases as is shown in Figure 4.



	Throws Exception	Returns Special Value
Add	add	offer
Retrieve	element	peek
Retrieve & Remove	remove	poll

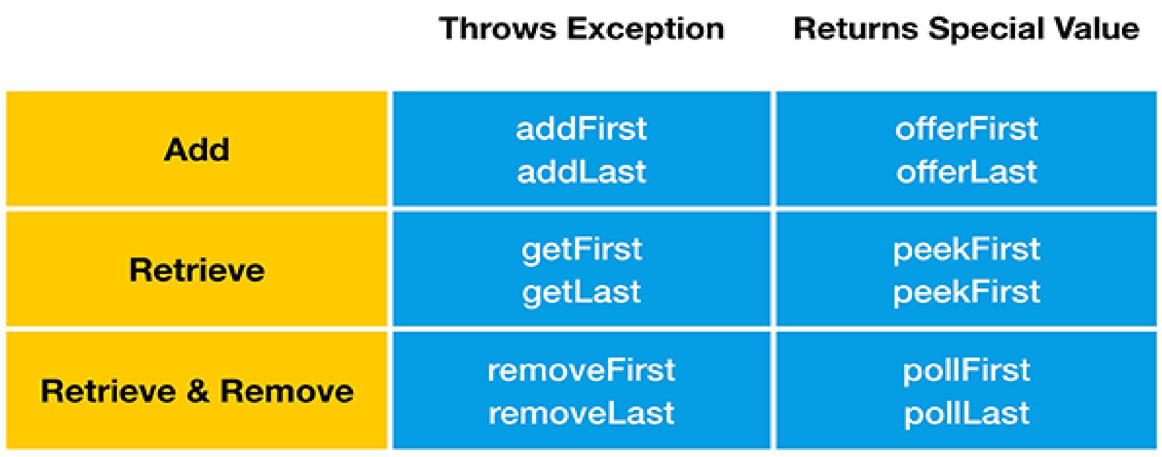
Figure 4

A Queue allows adding elements to its tail end.

In the case where add throws an Exception when the queue is full, `offer` returns false. LinkedList, like most Queue implementations, has a virtually unlimited capacity, so it will almost never be full. `ArrayBlockingQueue` on the other hand is one such queue implementation with a limited capacity.

Next up are `element` and `peek`. Both allow you to retrieve an element from the front of the queue without removing it. If the queue is empty, `element` throws an `Exception` while peek returns false. Finally, you can retrieve and remove an element from the front of the queue. If a queue is empty, `remove` throws an `Exception` while `poll`returns false.

Now we will look at some methods from the Deque interface as implemented by `LinkedList`. Deque is short for "double-ended queue", making it a queue that can be accessed from either end. Just like a queue, a deque allows **adding**, **retrieving**, and **retrieving and removing** an element. However, since it can be accessed from either end, the Queue methods we saw before now exist in two variations – one for the first and one for the last element of the deque, as shown in Figure 5.



	Throws Exception	Returns Special Value
Add	addFirst addLast	offerFirst offerLast
Retrieve	getFirst getLast	peekFirst peekFirst
Retrieve & Remove	removeFirst removeLast	pollFirst pollLast

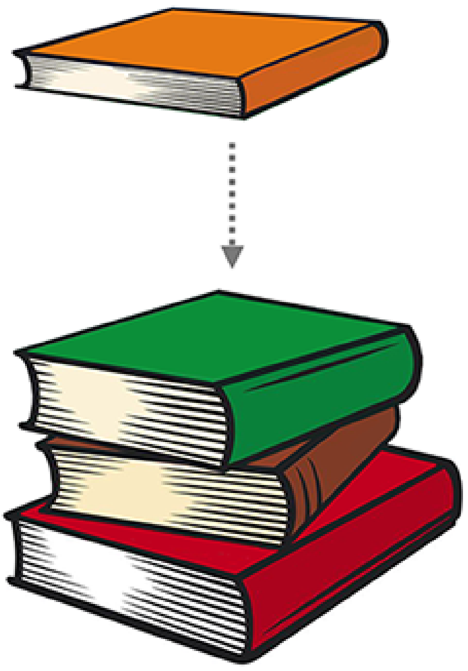
Figure 5

Again, let's look at this in more detail. You can add elements to both ends of the Deque. Just like the add method of the `Queue` interface, `addFirst` and `addLast` will throw an `Exception` when the `Deque` is full. `offerFirst` and `offerLast` will return false instead of throwing an `Exception`. Do keep in mind that `LinkedList` has an unlimited capacity so it will never get full. `LinkedBlockingDeque` on the other hand is a Deque implementation that may have a limited capacity.

You can retrieve elements from both ends of the Deque without removing them. `getFirst` and `getLast` throws an `Exception` when the queue is empty while `peekFirst` and `peekLast` returns false in this case. Finally, you can retrieve and remove elements from both ends of the Deque. `removeFirst` and `removeLast` throw an Exception when the queue is empty while `pollFirst` and `pollLast` return false in this case.

Stack

The Deque interface also supports methods of the `Stack` data structure, namely push, peek and pop. This allows us to use `java.util.LinkedList` as a `Stack`.



A stack is a very simple data structure that can only be accessed from the top. As an analogy, think of a stack of books:

`push` adds an element to the top of the stack, equivalent to the `addFirst` method. peek retrieves but does not remove an element from the top of the stack, just like the `peekFirst` method. `pop` retrieves and removes an element from the top of the stack, same with the `removeFirst` method.

Code Examples

We will be looking at some code snippets with a `LinkedList` instance. At this point, it should be mentioned that we have to import classes like `java.util.Deque`, `java.util.LinkedList` and `java.util.concurrent.LinkedBlockingQueue` to compile these snippets.

Adding to Deques and Queues

add, addLast, and addFirst

First off is a Deque reference variable, deque. Note the use of the **diamond operator, <>,** introduced with Java 7. The operator spares us the trouble of writing `String` again to the right of `LinkedList`. Also note the use of the Deque interface on line 1. This makes it so that we only get to use the methods of `LinkedList` which implement Deque. Since `LinkedList` implements other interfaces like the Collection and List interfaces, you might also want to review `ArrayList` which also implements them, by going over to our previous article on it. Going back to our example, calling `deque.add` on strings "1", "2" and "3" gives us a Deque that contains the elements `[1, 2, 3]` in that order.

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3"); // deque: [1, 2, 3]
```

```
Queue<String> queue = new LinkedList<>();
queue.add("1");
queue.add("2");
queue.add("3"); // queue: [1, 2, 3]
```

```
Deque<String> deque = new LinkedList<>();
deque.addLast("1");
deque.addLast("2");
deque.addLast("3"); // deque: [1, 2, 3]
```

Replacing deque with the Queue reference variable, queue, gives us the same outcome. Turning our Queue back into a Deque, we get to use `addLast` to add elements to the end of deque. Sure enough, we get the same outcome as in the previous two examples.

Starting over with an empty deque, we can use `addFirst` to add elements in reverse order to `addLast`, giving us 3, 2, 1. We can mix it up by using both addFirst and addLast in one place, but this can get really confusing so we should try to avoid doing so.

```
Deque<String> deque = new LinkedList<>();
deque.addFirst("1");
deque.addFirst("2");
deque.addFirst("3"); // deque: [3, 2, 1]
```

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.addLast("2");
deque.addFirst("3"); // deque: [3, 1, 2]
```

offer, offerFirst and offerLast

Analogous to the three add methods, we have three offer methods which don't differ so much from the former when it comes to LinkedLists. With LinkedBlockingDeque however, because we can limit the maximum number of elements in the queue, adding one over the maximum, we would expect something to happen. As it turns out, the offer group of methods would just return a boolean indicating whether the element was accepted or not. In the case where we add another element to a full deque, our offer methods would return false.

```
Deque<String> deque = new
LinkedBlockingQueue<>(2);
deque.offer("1");
deque.offerFirst("2");
boolean wasAdded = deque.offerLast("3");
// wasAdded: false
// deque: [1, 2]
```

What happens if instead we use an add method on a full LinkedBlockingDeque? Our program would actually throw an IllegalStateException. In normal circumstances, the same exception is not thrown with a LinkedList as it is a virtually unlimited Queue and Deque implementation.

```
Deque<String> deque = new LinkedBlockingQueue<>(2);
deque.offer("1");
deque.offerFirst("2");
deque.add("3"); // throws IllegalStateException
```

On a final note, another difference between the add and offer methods is that add returns void while offer returns a boolean. This is probably a minor difference, because it would only matter when a Queue or Deque implementation has a limited capacity.

Reading from Deques and Queues

element

A general-purpose function for reading from deques and queues is element. As our snippet shows, we always retrieve the leftmost element and no elements are removed from deque afterwards.

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");

String element = deque.element(); // element: 1
// deque: [1, 2, 3]
```

As for explicitly retrieving an element from the front or the back of our deque, we actually use **getFirst** and **getLast**.

getFirst and getLast

To explicitly retrieve an element from the front of a deque, we use `getFirst`. In our case, it works exactly the same as `element` from our last example.

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");

String first = deque.getFirst(); // first: 1
// deque: [1, 2, 3]
```

Conversely, we use `getLast` to explicitly retrieve an element from the back of a deque.

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");

String last = deque.getLast(); // last = 3
// deque: [1, 2, 3]
```

peekFirst and peekLast

As with add and `offer`, the methods for reading from a deque are very similar. The difference shows with an empty deque or queue.

```
Deque<String> deque = new LinkedList<>();
String first = deque.peekFirst(); // first = null
// deque: []
```

```
Deque<String> deque = new LinkedList<>();
String last = deque.peekLast(); // last = null
// deque: []
```

With `element`, `getFirst` and `getLast`, an empty deque gets us a NoSuchElementException.

```
Deque<String> deque = new LinkedList<>();
String element = deque.element();  // throws NoSuchElementException
```

```
Deque<String> deque = new LinkedList<>();
String element = deque.getFirst(); // throws NoSuchElementException
```

```
Deque<String> deque = new LinkedList<>();
String element = deque.getLast();  // throws NoSuchElementException
```

The question now becomes which methods to use for which cases. Whenever we expect to read off an empty deque, we would probably go for **peek** methods. For limited deques and queues, we would probably use the **offer** methods. Conversely, when we consider not being able to add to deques or queues an exceptional case, we would use the **add** methods. The Queue and Deque interface is quite flexible, providing us with different types of behaviors.

Removing from Deques and Queues

Starting with empty deques, removing from them causes our code to throw a NoSuchElementException.

```
Deque<String> deque = new LinkedList<>();
deque.remove(); // throws NoSuchElementException
```

Using poll instead, we get null as its return value. The same is true for pollFirst and pollLast.

```
Deque<String> deque = new LinkedList<>();
String element = deque.poll(); // element = null
element = deque.pollFirst();   // element = null
element = deque.pollLast();    // element = null
```

Of course, the remove and poll methods work similarly to the previous methods, as expected.

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");

String first = deque.removeFirst(); // first = 1, deque: [2, 3]
String last = deque.removeLast(); // last = 3, deque: [2]
String element = deque.remove(); // element = 2, deque: []
```

In the example above, note that since a LinkedList allows duplicates, we could have added, say 1, three times. We purposely added 1, 2, and 3 so we actually see which element was removed. removeFirst retrieves and removes the leftmost element and so too for the rightmost element with removeLast.

Stack

Lastly, we cover stacks and how they relate to the previous topics. The methods that define stack behavior are actually also part of the Deque interface. In fact, there is no Stack interface. There is a Stack class which extends Vector but as per an earlier article, the Vector class is to be avoided, which means that Stack should be avoided too. Being 20 years old, the performance side is quite suboptimal. We can use something as simple as an array or an ArrayList of course, but instead we will look into Deque's stack methods.

```
Deque<String> stack = new LinkedList<>();
stack.push("redBook");
stack.push("brownBook"); // stack: [brownBook, redBook]

String top = stack.peek(); // top: brownBook, stack: [brownBook, redBook]
top = stack.pop(); // top = brownBook, stack: [redBook]
top = stack.pop(); // top = redBook, stack: []
stack.pop(); // throws NoSuchElementException
```

To add an element to a stack, we call the method `push`. Since stacks are first-in, last-out data structures, when we add, say "`redbook`" and then "`brownbook`" to our stack, we can think of the brown book being on top of the red one.

If we want to just retrieve or remove a book, the brown book gets taken because we can only do things to the top of a stack. In our case, "`brownBook`" is what `stack.peek()` returns as the top element. Note that `LinkedList` stores the elements starting from top to bottom in a way in which when printed out would appear from left to right.

Afterwards, we **pop** elements off the stack one by one and see that "`brownBook`" is the first to be taken out, reducing the stack's size to one, followed by "`redBook`", making the stack empty. Finally, removing yet again, this time from an empty stack, would cause our code to throw a NoSuchElementException.

```
Deque<String> stack = new LinkedList<>();
String top = stack.peek(); // top = null, stack: []
```

Finally, if we instead called `peek` on an empty stack, we would be returned a `null` value for `top` with no exception thrown. The stack remains empty all the same.