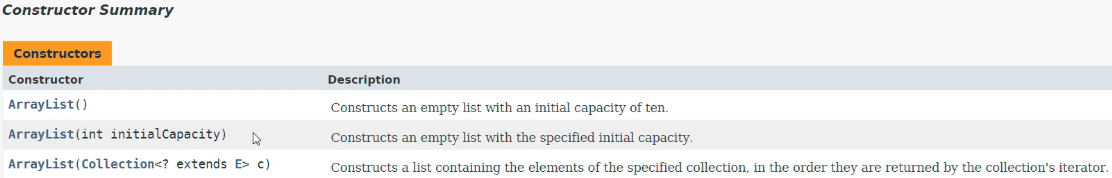
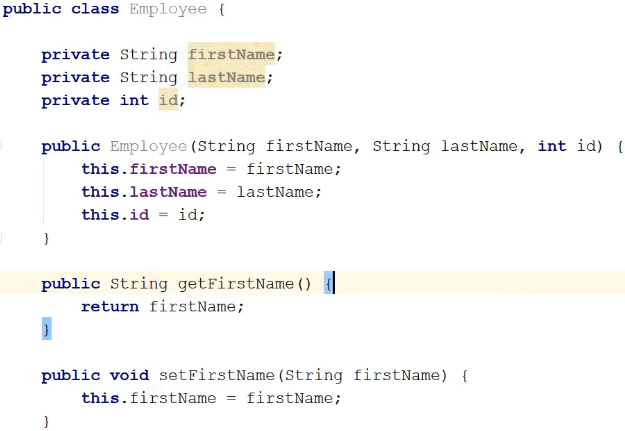
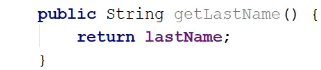
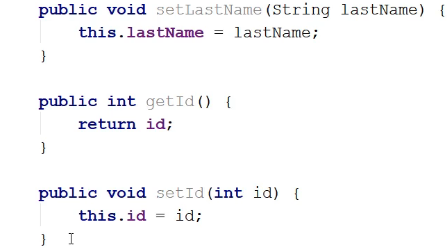
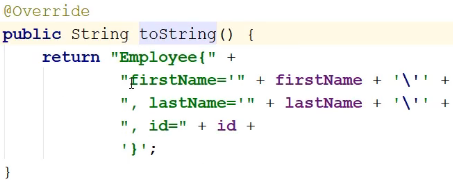
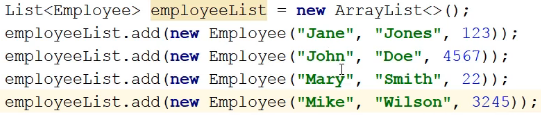
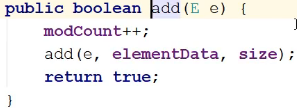
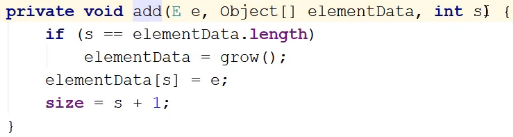
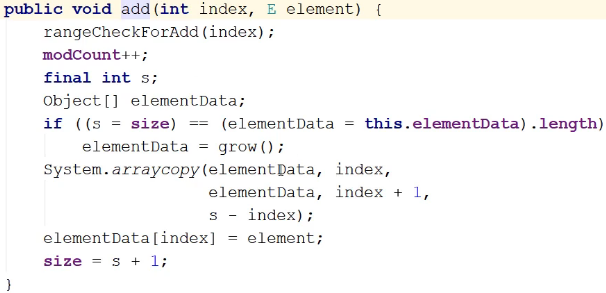
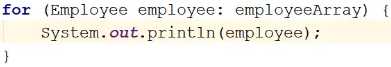
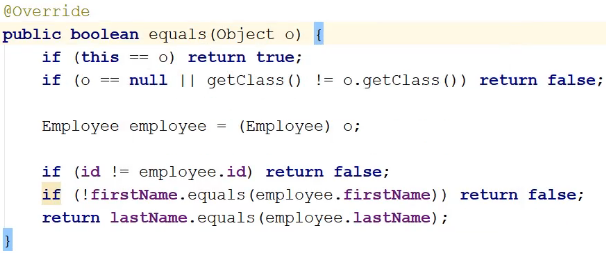
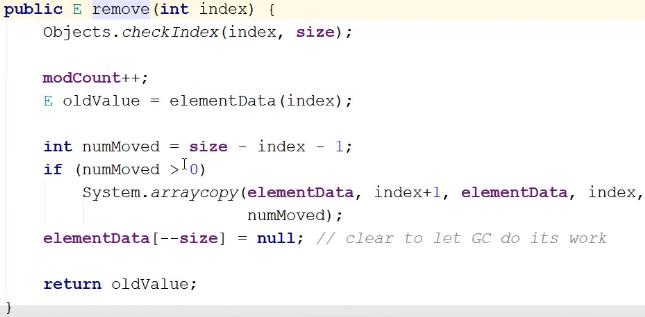
**Introduction to Lists**  
\* The List is an abstract data type.  
\* **List isn’t a concrete data structure, it’s an abstract data type**.  
\* **When it comes to abstract data types, normally there’s an interface involved**.  
\* In Java, the Lists that we’re going to look at in this section all implement the **java.util.List** **interface**.  
\* **Classes that implement the List interface represent an ordered collection also known as a sequence**.  
\* There are quite a few classes that implement List.  
\* We’re going to look at:  
**ArrayList  
LinkedList  
Vector**  
\* There’s also an AbstractList and AbstractSequentialList - if you want to implement the List interface, you want to create a custom implementation, rather than starting with List itself and implementing that, it’s a good idea to instead extend AbstractList or AbstractSequentialList because they kind of give you a head start and all you have to do is override the methods that you specifically want to implement, if you want custom behavior.  
\* So if you’re going to implement a custom list, it’s a good idea to extend one of these two classes.  
\* Let’s look at a few of List methods:  
**add()  
contains()  
get()  
indexOf()  
isEmpty()  
remove()  
size()  
toArray()** => get the array that’s backing a List.  
\* **When dealing with Lists, we’re dealing with an ordered sequence, so the data is organized sequentially just as it is in an array and we’re going to see that a popular way to implement the List interface is to use an Array**.

**Abstract Data Types**  
\* Abstract data type isn’t a concrete data structure in the sense that an array is.  
=> **Doesn’t dictate how the data is organized**  
=> **Dictates the operations you can perform**  
=> **Concrete data structure is usually a concrete class**  
=> **Abstract data type is usually an interface**  
\* With arrays, we know that we have to store all the items as one contiguous block and arrays also dictate that every item in the array has to occupy the same amount of memory.  
\* So arrays are telling us how the data has to be stored.  
=> Lists don’t do that. Lists are more of a conceptual idea and they dictate the operations we can perform on the data set, not the items themselves. An abstract data type doesn’t do something like say: you can only multiply these items together - it’s more that it dictates how we can access the items.  
=> Can we do random access?  
=> Can we get to the first item that we added?  
=> Can we get to the last item that we added?  
\* **And so, an abstract data type is more about behavior and what operations you can do**.  
\* In Java, a concrete data structure like an array is usually a Class.  
\* Arrays might actually be an exception to that.  
\* **When it comes to an abstract data type, normally those are interfaces, they’re not telling you how to store the data, they’re specifying behavior and that’s commonly done with an interface**.  
\* And so basically, any data structure can be used to implement an abstract data type. As long as you have a class that implements the interface for the abstract data type, any class can behave like that abstract data type.  
\* So in the case of Lists, **any class that implements the List interface, is a List**.  
\* And so you could have a Class that uses an array for the List. **If that class implements the List interface, it can be treated like a List**.  
<https://docs.oracle.com/javase/9/docs/api/java/util/List.html>  
\* Lists say for example: you can add an item, you can remove an item, you can get the index of an item, etc.  
\* They DON’T say: and this is the way you have to store the items in the List.

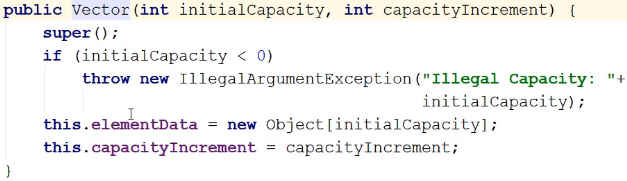
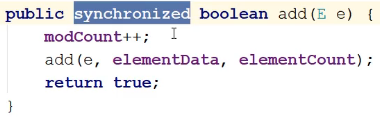
**ArrayLists**  
**ArrayList** => pretty much the go-to class when it comes to storing a collection of objects that you’ll want to iterate over sequentially.  
<https://docs.oracle.com/javase/9/docs/api/java/util/ArrayList.html>  
\* The documentation says: It’s a **resizable-array implementation of the List interface**, meaning that the data in the List is being stored in an array. This array is called the **Backing Array**.  
\* This tells us a few things based on what we’ve learned about arrays:  
=> **If we know the position of an item in the List, accessing it will be efficient, O(n) - constant**.  
=> **ArrayLists are great if all you’re going to do is iterate over them because in that case you’re always going from index 0 to the end of the List, so you know the indices of the items that you want to retrieve**.  
=> But if you want to add a lot of items to an existing List, this will be slow if the size of the List isn’t large enough to accommodate the new items - if the backing array is already full, the implementation of the ArrayList class will have to resize that backing array.  
=> Also removing items will be slow because we’re going to have to shift the remaining items to remove any empty space.  
\* So if you’re going to be adding a lot of items into an ArrayList, you want to have some idea of how many items are going to ultimately be in the List, so you can create an ArrayList instance with a capacity that will accommodate all the items.  
  
\* When passing the initialCapacity, the backing array will be of length initialCapacity.  
\* When it comes to ArrayLists, it’s important to understand the difference between size and capacity.  
**capacity** => the maximum number of items that the List can hold before it’s going to have to be resized (the backing array).  
**size** => the number of items that are actually in the List.  
\* So if you create an ArrayList with initialCapacity of 20 and add 3 items, then the capacity is 20 and the size is 3.  
\* If you don’t pass the capacity to the ArrayList, you can see that you get initialCapacity of 10.  
\* So if you know you’re going to have more than 10 items, you’re better off specifying the capacity.  
\* Let’s now play around with ArrayLists in the IDE.  
  
  
  
\* Finally, I’d like a **toString()** method generated, I want to override the default because when we print an Employee instance, I’d like to see firstName, lastName and id, **I don’t just want to see the object reference**.  
  
\* Now let’s go back to our Main method.  
**List => the reason I’m using List rather than ArrayList is that by using List, if I decide later that I want to use a different type of List, it’s easy to change the specific implementation of List that I’m using.  
ArrayList<>() => the <> is a Diamond type.  
add()**  
  
\* What’s happening under the covers when we call the ArrayList constructor is it’s creating a backing array of length 10.

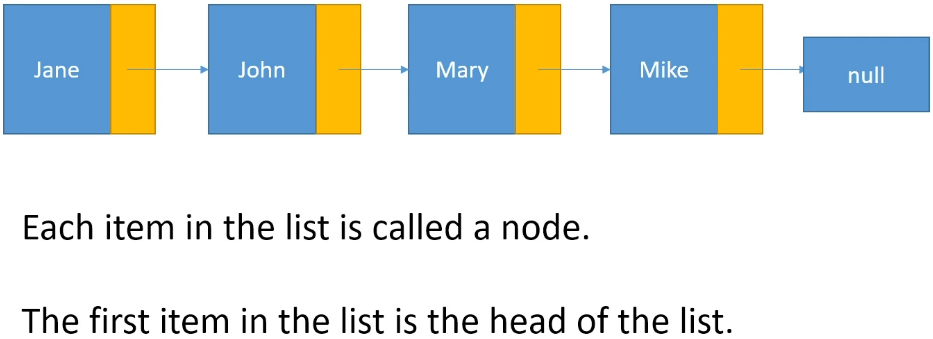
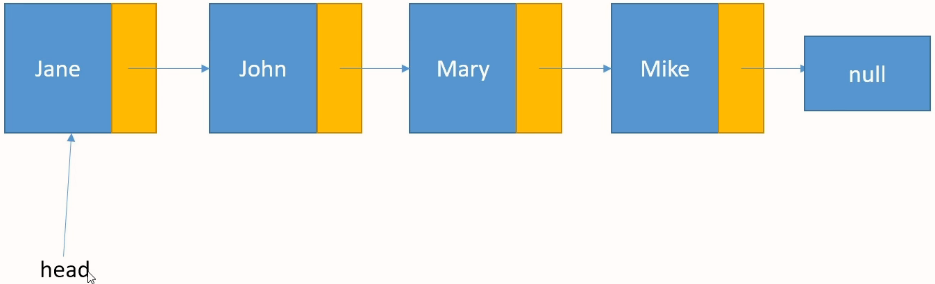
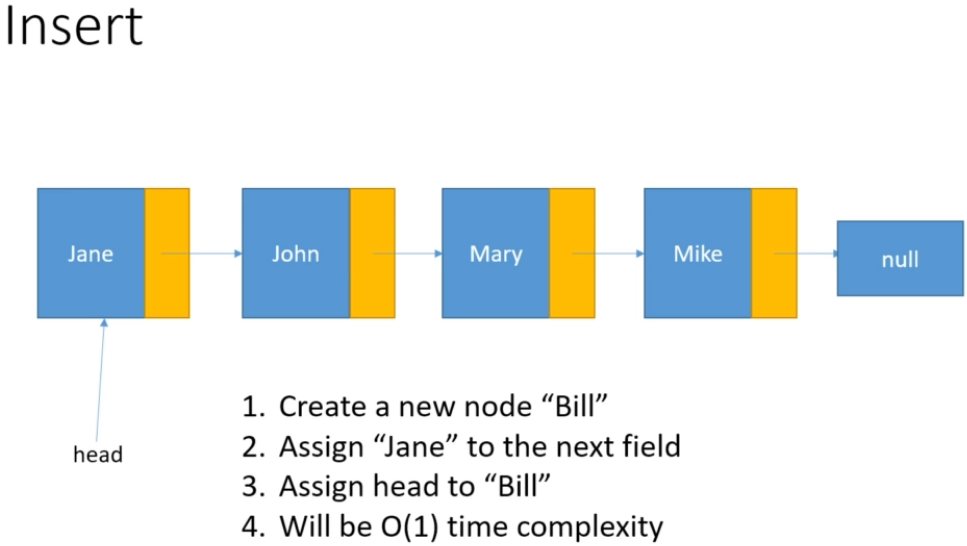
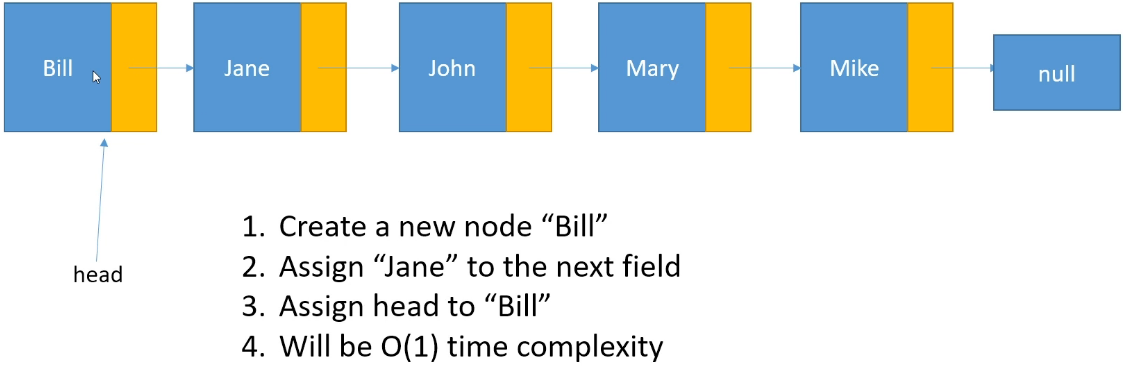
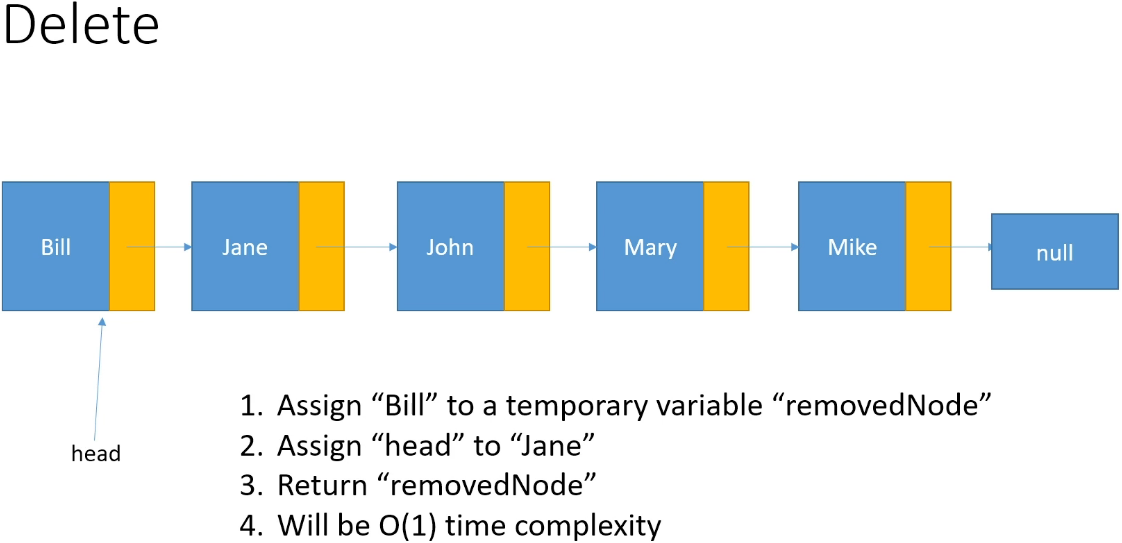
\* Let’s now Go To the declaration of the add() method.  
  
\* This is in the interface, I want to go to the implementation, we can click on the green I icon on the left and select the **ArrayList.add(E) (java.util)**.  
\* Here we are in the actual code.  
  
\* Now let’s go to this add() declaration.  
  
=> The first thing it does is checks to see whether the backing array is full. If it’s full, it needs to grow() the array, it has to resize it.  
\* So an ArrayList is backed by an array, so why don’t we just use an array?  
=> Because ArrayList has a bunch of methods that let you work with the List items and so the code to do this has been written for you, you don’t have to directly work on the array.  
=> Also, ArrayList implements the List interface and so you can swap to another type of List that uses a List interface without too much disruption of the existing code.  
**forEach()**  
\* I could’ve just used a loop and loop through the listbut this is more concise.  
  
**get() => it’s done in a constant time, because an array is backing the List, random access like this where we just provide an index, the time complexity doesn’t depend on the size of the List.**  
  
=> **ArrayLists are great when you want to load a bunch of data and then access it like this**.  
**isEmpty()**  
  
\* We can replace an item at a specific position with another item.  
**set()**  
  
\* If you want to add() an item to the List, it’s always added to the end of the List.  
\* If you want to replace an item in the List, you use the set() method.  
\* **Random access is O(1) so set() will also happen in constant time as long as you provide the index**.  
**size()**  
  
\* If you want to add an item at a specific position, you use the add() method and provide it with more parameters:  
  
\* You’re providing an index but because in this case you’re sticking him in the middle of values, some values are going to have to be shifted up, so the **worst case for this would be O(n)** because the worst case would be putting the item right at the beginning of the array.

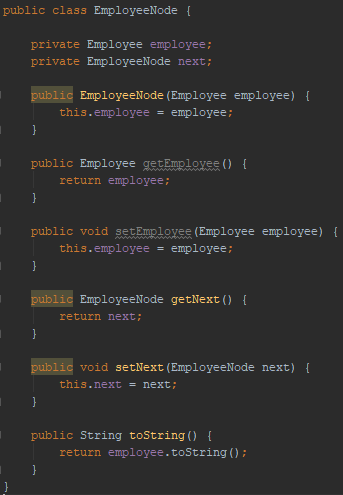
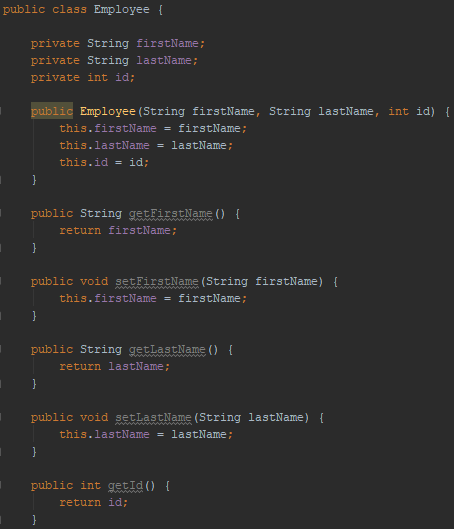
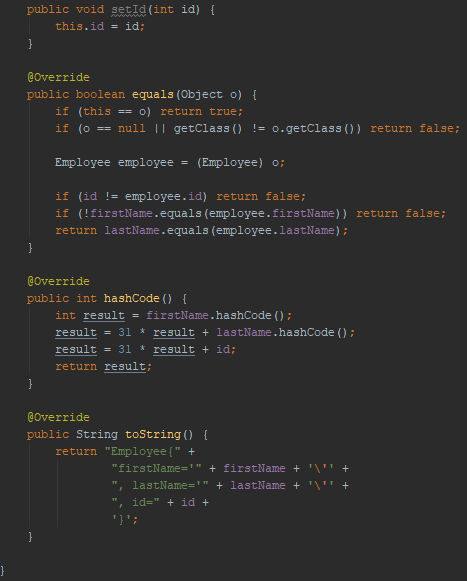
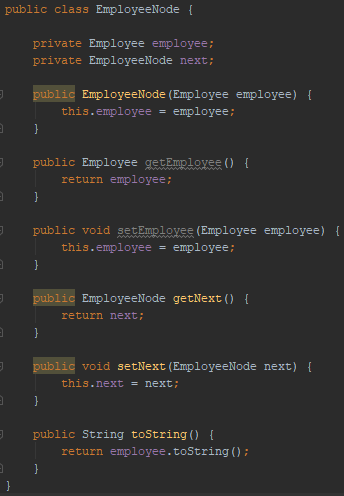
  
\* If we want to get that backing array, the array thata’s holding the Employees, we’re can use the **toArray()** method but **we’re not going to get an Employee[] array**, **we’ll get an Object[] array**.  
  
\* **If we want an Employee[] array**, we can get it but then we have to tell the compiler that that’s what we want.  
=> To do that, we pass an array of the type we want.  
  
\* We can now change it to Employee[]:  
  
  
\* If you want to know if the List contains an instance:  
**contains()**  
  
=> We get false.  
=> The reason you get false is because **we haven’t implemented the equals()** method in the Employee class.  
=> **Because there’s no equals() method, what this method is doing is actually checking to see if they’re the exact same instance. That’s what the default equals() method does - the one that’s all the way back in Object class**.  
\* The 2 items we tried to compare are 2 distinct instances, they’re structurally the same.  
**equals()**  
=> **Whenever you’re overriding the equals() you should be overriding the hashCode() as well**.  


\* **Now that we have the equals() method, we can make use of the indexOf() method**.  
**indexOf()** => this will look up an Employee in the List and tell us what index that item is occupying.  
  
\* **The contains() and indexOf() methods are only as good as the search algorithm being used to search the array**.  
\* In other words, when we’re looking something up in the List, **the method that we’re using has to search the List and the way it’s searching the List will determine whether the contains() and indexOf() method are fast or slow**.  
**remove() => you can pass the index or an instance of the item, but the index is faster.**  
\* Because we’re dealing with an array in the background, remove() can be expensive because elements after the one we removed have to be shifted down 1 position.  
  
\* There’s a System.arraycopy() and that’s what’s doing the shifting, if it has to shift elements down to occupy the empty space left by the 1 that we removed.  
\* **So ArrayList is good for random access if you have the index and it’s good for iterating over the items in the List**.  
\* **It’s not so good for inserting items into the List in any position other than the end**.  
\* **It’s not so good for deletions and removals**.  
\* **It’s not so good for accessing an item in the List when you don’t have its index**.  
\* Remember, it’s **backed by an array**.  
\* **So it has the asme problems and advantages that arrays have**.

**Vectors**  
**Vector** => **essentially a Thread-safe ArrayList**.  
\* But Vector actually came first, since JDK 1.0 while ArrayList was added in JDK 1.2.  
<https://docs.oracle.com/javase/9/docs/api/java/util/Vector.html>  
\* **Vector: unlike the new collection implementations, Vector is synchronized**.  
\* Why was ArrayList added to the JDK?  
\* **ArrayList: this implementation is not synchronized**.  
\* So Vector is Thread-safe meaning it’s okay to use it from different Threads without you having to synchronize the code.  
\* ArrayList is not.  
\* So if you **only read an ArrayList**, then it’s **safe to use it from multiple Threads**, no data will be clobbered, but if you use ArrayList with **multiple Threads and one or more of those Threads is writing to the ArrayList by adding, deleting, setting or changing objects** in the List, then you **could run into a conflict**, you could have Threading conflicts. And so in that case, you’d want to use Vector rather than ArrayList.

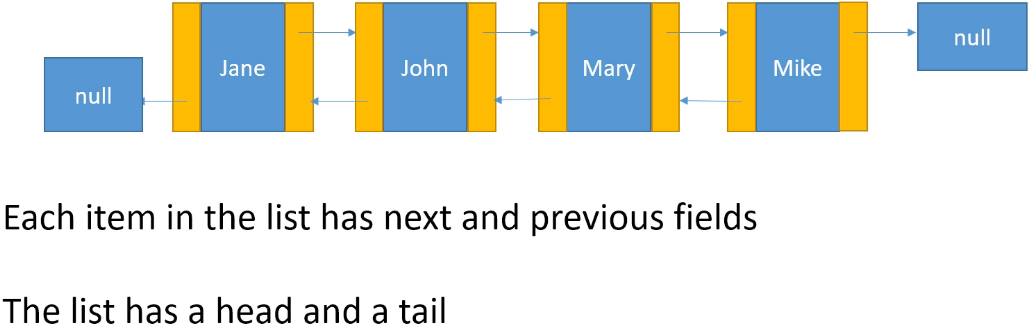
\* The reason ArrayList came along is that **synchronization has an overhead involved**. So it slows things down.  
\* If you need Thread-safety, synchronization => Vector.  
\* If you don’t need Thread-safety, synchronization => ArrayList.  
\* **Since we used a List** and Vector also implements a List, all we have to do to change it is to use a Vector:  
  
\* That’s why rather than using a specific implementation in the declaration, it’s always better to use the interface that the implementation is using because then you can switch out to another class that implements the same interface without having to change any code.  
  
  
\* **That’s it, I just wanted to make sure that we covered the 2 most popular classes for in the JDK when it comes to just having a simple sequential List**.

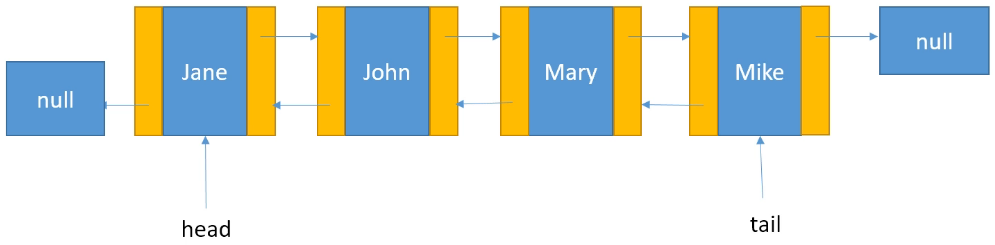
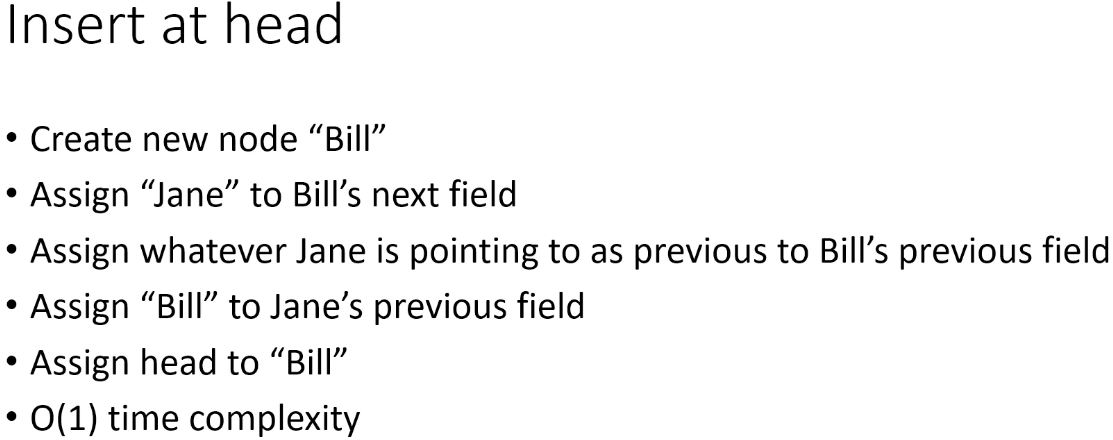
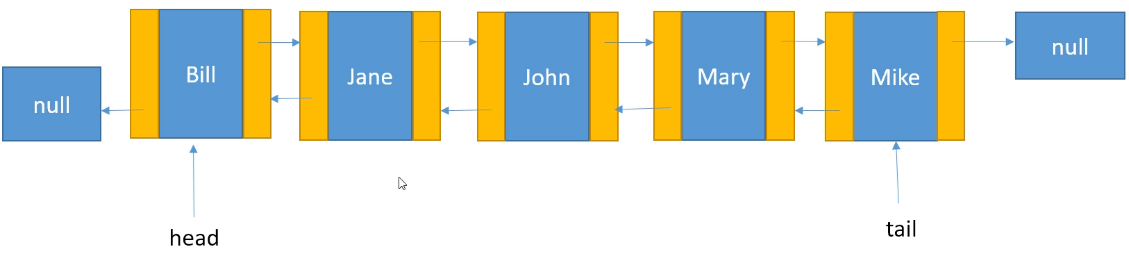
**Singly Linked Lists (Theory)**  
\* Linked List is a data structure.  
\* It’s a sequential list of objects.  
\* This time, arrays aren’t involved.  
  
\* In a Linked List, **each item in the list is aware of another item in the list**.  
\* Each item in the list contains a link to the next item in the list.  
\* With an array, each item in the list is completely unaware of other items in the array.  
\* Items in a Linked List know which item comes after them and that means that we have to store some extra information with each item.  
\* We have to store the value and a reference to the next value in the list.  
\* Each of the items is referred to as a Node.  
\* The first item in the list is the Head of the list.  
\* The last item in the list always points to null.  
\* So we’ll have a Node class that contains a field for whatever data you’re holding in the Node and then we’ll have a next field and that will be of type Node because each Node points to the Node that comes after it.  
\* In Java, you wouldn’t implement a Linked List yourself, there’s a LinkedList class but to help us understand what a Linked List is and what its advantages are, we will code a simple implementation.  
\* If you have a reference to the head, you can traverse the entire list - going next until you hit null.  
  
\* For a Linked List, the only thing you have to store is a reference to the head or the first node in the list and from that, you can get to every item in the list.  
  
**O(1)**  
\* **When it comes to Linked Lists, you always insert a new element at the front of the list**.  
\* It’s because we only ever store a reference to the first element and so if we wanted to insert an item anywhere other than the front, then we’d have to traverse the list to get there and   
**one of the advantages of using a Linked List is that if you insert items at the front of the list, you can do it in constant time complexity** because the steps you have to do don’t depend on the number of items in the list, you’re always going to do the same number of steps.  
  
  
**O(1)**  
\* Once again, we’ll want to delete from the front of the list, otherwise we’d have to traverse the list.  
\* In the implementation I’ll show you, when we do a deletion, we return the node that we deleted.  
\* Moving head to Jane effectively removes Bill from the list because for a Linked List, the only sort of information that we’re holding is this head field.  
\* After the deletion, Bill will still be pointing to Jane but there’s no way for us to reach him from the head of the list.  
\* And if we wanted to do a clean-out, we could set Bill’s next field to null if we wanted to.  
\* After arrays, this is the second data structure we’re looking at.  
\* Linked Lists differ from arrays in that as long as you’re inserting and deleting from the front of the list, the insertions and deletions are done in constant time. And that’s because there’s no shifting involved.  
\* This type of list is called a **Singly Linked List** because we have 1 link between every node.  
\* When you’re working with a Singly Linked List, you want to insert and delete items at the front of the list because you only have a reference to the head of the list and so if you want to insert and delete items anywhere else, you have to start at head and you’ve got to traverse the entire list to find what you’re looking for.

**Singly Linked Lists (Implementation)**  
**(Implemented my own version of SinglyLinkedList before watching this video)**  
\* By “contains a link”, we mean that it stored the object reference of the next node.  
\* I’m going to call this EmployeeNode rather than just plain old Node because I’m not going to use generics to write this LinkedList, I want us to just focus on the Linked List implementation. And on top of that, you’re not going to write your own Linked Link, you’ll end up using the one in the JDK and even if you did write a Linked List class for your application, you’d probably make it specific to the type of data you’re dealing with.  
=> The only reason that you would need to use generics is if you’re going to write a class that’s going to be released publicly and so that many, many applications are going to use it and in that case, you’d want to use generics because you’d want it to be usable with a variety of object types.  
  
\* **We don’t have to set the next to null in the constructor because that’s the default value for object fields**.  
   
  
  
\* In the toString() method in Node, we call the Employee toString() method so that when we print a Node, what we’ll actually be printing is the result of the Employee toString() method.  
\* **We could write a method that adds employees to the end of the list or that looks for a specific employee in the list and adds an employee after that employee or before the employee, but those methods will be on the order of O(n)**. That’s because then we have to traverse the list and the worst case will always be that we have to traverse the entire list.

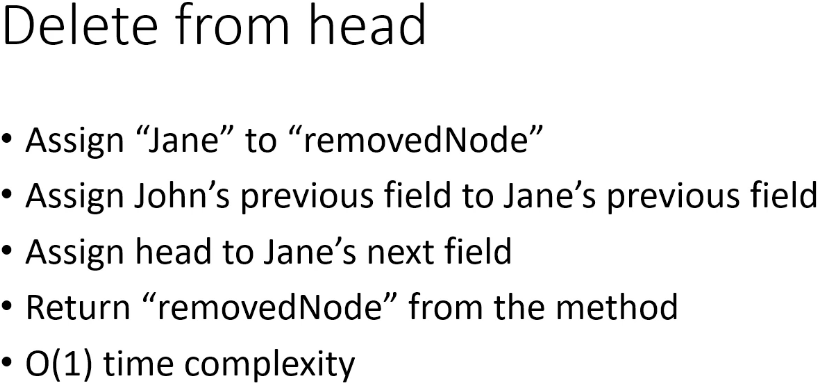
\* **Some implementations of Linked List will have a pointer to the tail of the list, the last node in the list, but that’s not really a true SinglyLinkedList, it’s a variation on them**. But you could do that, if you thought you were constantly going to be wanting to add items to the end of the list.  
\* **Singly Linked List is best used when you want to insert / remove items from the front of the list**.  
\* **Linked Lists can continue to grow without having to be resized**.  
\* You’re pretty much only bounded by the memory you have.  
\* **One disadvantage of a Linked List is that you have to store that extra field with every value.  
\* You don’t have to do that with arrays**.  
=> **So if memory is really tight, that could be one disadvantage to using a Linked List**.  
=> **If you want to do a bunch of random accessed, then a Linked List would be a bad choice**.  
=> **If you want to load a bunch of data into the list and you’re always going to be most interested in whatever’s at the front of the Linked List, then that could be a really good choice**.  
\* As usual, it’s going to depend on your application, the platform you’re running on, what the application is going to want to do with the data etc.  
\* Let’s say we wanted to know how many items are in the Linked List.  
=> We could traverse the list and count how many items there are.  
=> But another way to do it would be to just keep a running count of how many nodes are in the list.

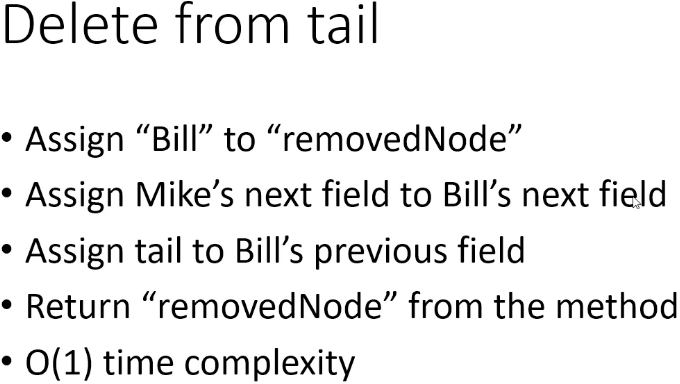
**Doubly Linked Lists (Theory)**

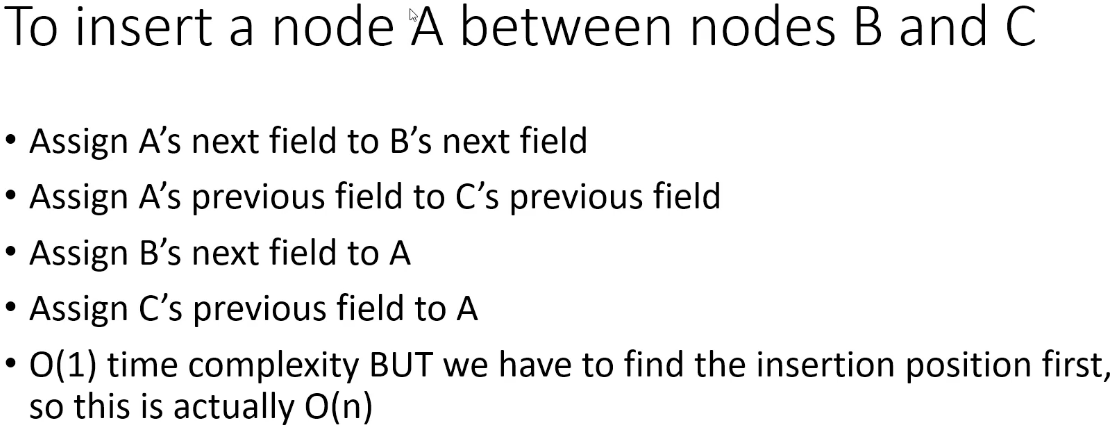


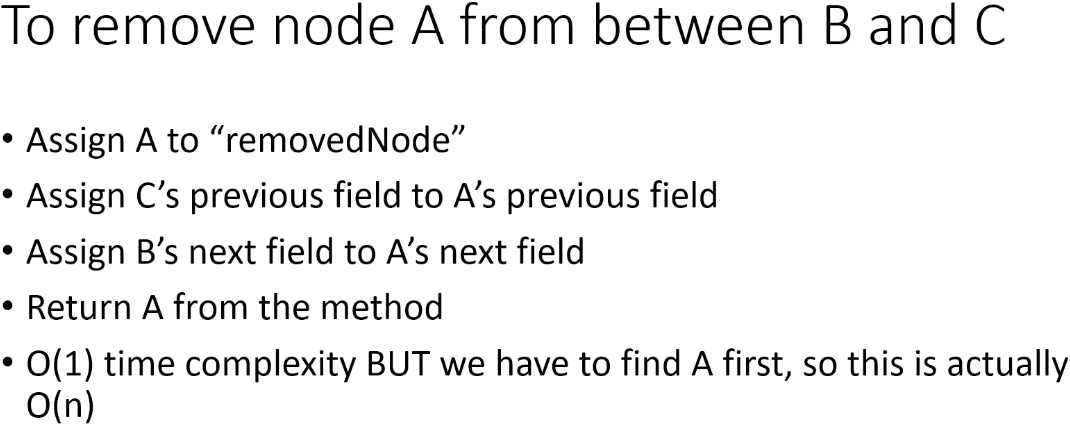
\* **Double Linked List** / **Doubly Linked List**   
\* **We have nodes, head and tail**.  
\* **Each node in the list points to the next item in the list and to the previous item in the list**.  
\* For head, the previous points to null.  
\* For tail, the next points to null.  
  
\* **We can traverse the list from head to tail or from tail to head**.  
=> **So this time if we want to insert/remove a node from the end of the list ,we can do it in O(1).**  
\* **So the advantage of using a Doubly Linked List is that you can work with the node at the front of the list or the node at the end of the list, in constant time**.  
\* But if you want to work with the nodes in the middle of the list, you’re going to have the same problem that a SinglyLinkedList had, you’re going to have to traverse from the head or from the tail, find the node you want to work with and in the worst case, that could be a linear operation O(n).  
  
  
\* **We go through the same steps if we were to insert a node somewhere else, in that case we would have more references to update but we would go through a bunch of similar steps**. We would set Bill’s next and prev, then we would have to update the next of the node before and the previous of the node after.

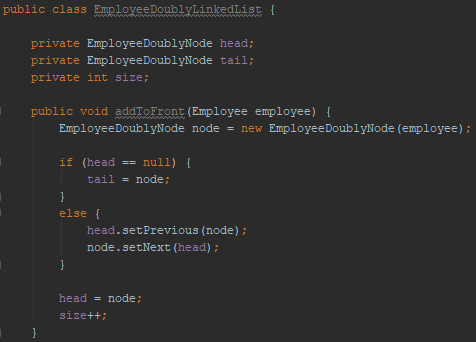
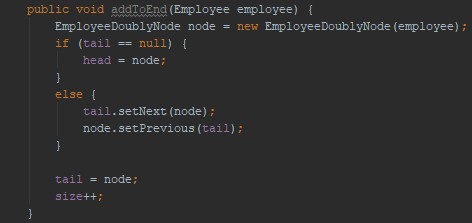
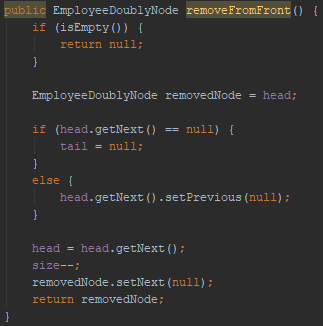
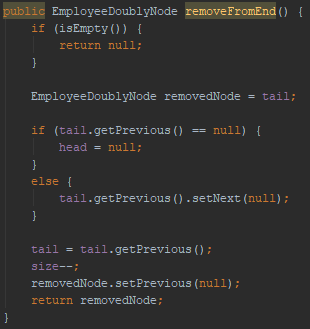
\* But as I said, when you’re working with Linked Lists, you mainly want to be focusing on working with items that are at the front of the list, or in the case of Doubly Linked Lists, at the tail of the list as well.  
\* If you start playing around with items in the middle, you’re going to lose the advantage of a Linked List, which is that inserting/deleting from the front (and end in case of Doubly Linked List) can be done in constant time O(1).  

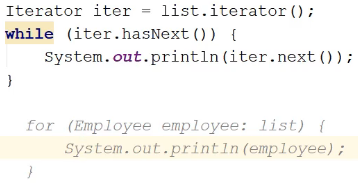



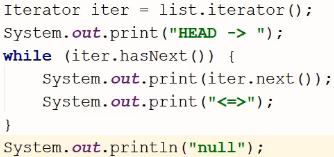
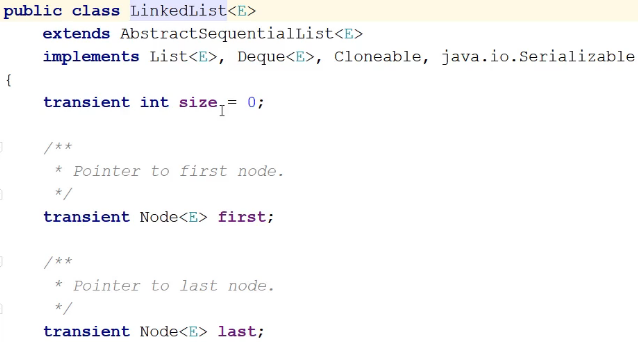
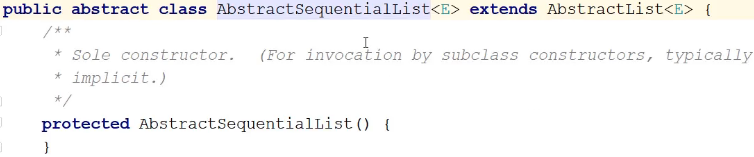
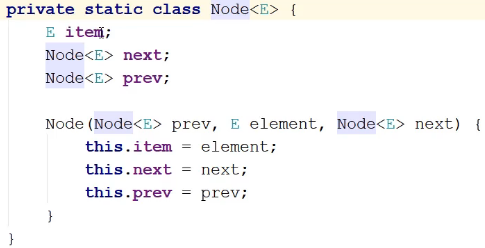


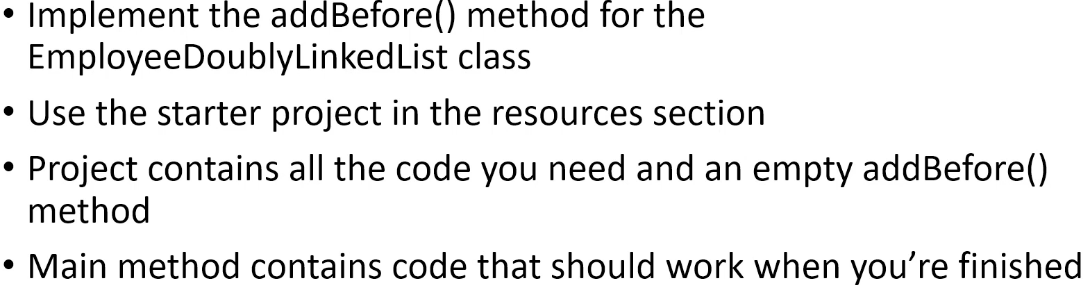


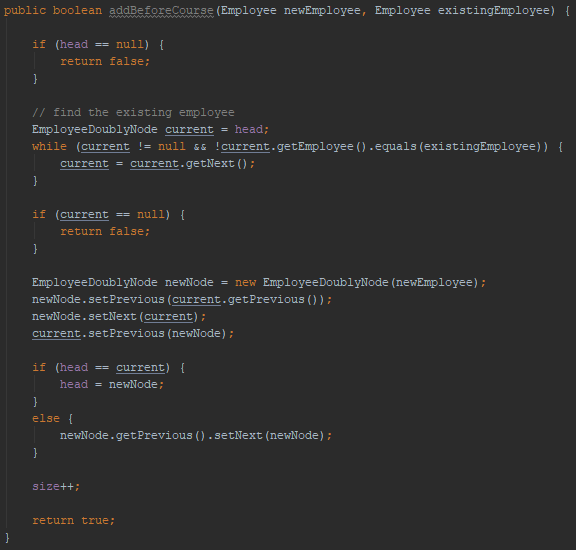
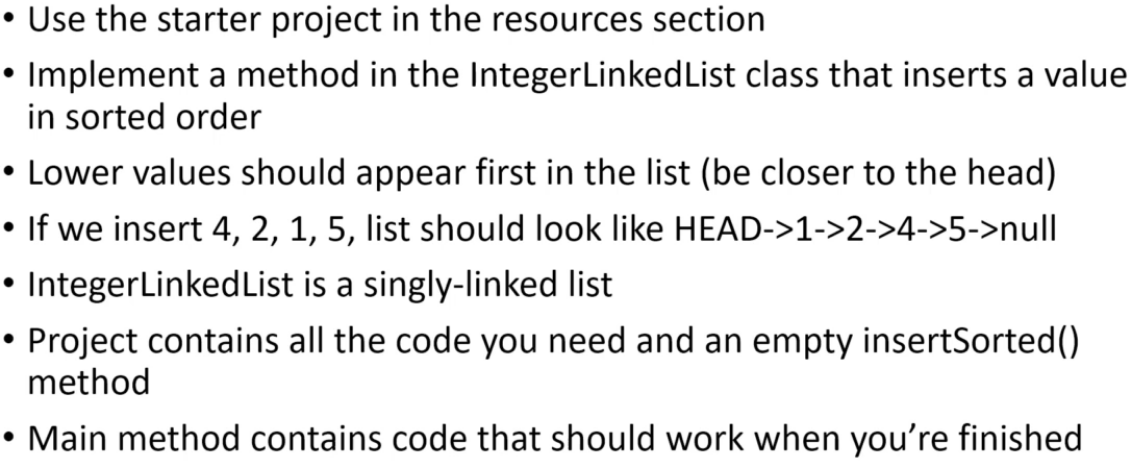


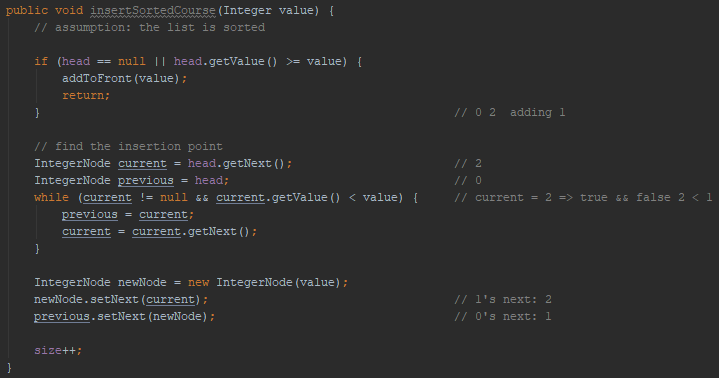
**Doubly Linked Lists (Implementation)**  
**(Implemented my own version of SinglyLinkedList before watching this video, including:  
insertAtHead, insertAtTail  
deleteFromHead, deleteFromTail  
insertBetweenAB  
deleteBetweenAB)**  
\* In Node, we need to add a field for a link to the previous item.  
\* In DoublyLinkedList, we need to add a field for the tail  
  
\* We don’t need to set the node’s previous field to null because that’s what it is by default.  
  
\* We don’t need to set the node’s next field to null because that’s what it is by default.  
  
\* If we want, we can clean up the references - set the deleted node’s next link to null.  
  
  
\* That’s it.  
\* The JDK has a LinkedList class, so if you want to use a LinkedList, you’re probably going to use the class in the JDK.

**The JDK LinkedList Class**  
\* If you want to use a LinkedList in Java, you’re probably going to use the LinkedList class in the JDK.  
<https://docs.oracle.com/javase/9/docs/api/java/util/LinkedList.html>  
\* It’s a **Doubly-linked list implementation** of the **List** and **Deque** **interfaces**.  
\* Because it implements the List interface, all the methods in the List interface are in the LinkedList class, so we have methods like   
**add()**  
**remove()**  
**indexOf()**  
**size()**  
**toArray()**  
  
=> **This class uses** **GENERICS**, **so you can use LinkedList class with any type of object**, including our employee object.  
\* **The LinkedList class has its own implementation of the Node class**.  
\* This class is **not synchronized**.  
=> So if you want to use a LinkedList instance from multiple Threads, you’ll have to synchronize the calls to any of the LinkedList methods yourself.  
**LinkedList<> x = new LinkedList<>()**  
\* The LinkedList class doesn’t have a handy print method, but **it does have an** **iterator**, so that’s how we’re going to print what’s in the list.  
**LinkedList** **addFirst() => adds to the front of the list**  
**Iterator**  
**LinkedList** **iterator()**  
**Iterator** **hasNext()**  
**Iterator** **next()**  
  
\* **We could also use a for loop to print out the list**.

\* We can print it in the same way we did before like this:  
  
\* Let’s say we want to add Bill to the end of the list.  
**LinkedList** **add()** => **adds an item to the end of the list - it’s the implementatin from List interface**.  
\* It’s important to read the description of the method you want to use.  
**LinkedList** **addLast()**  
  
\* What’s the underlying data structure being used for the LinkedList?  
\* For ArrayList and Vector it was an array.  
\* LinkedList extends AbstractSequentialList:  
  
=> This kind of provides a skeletal implementation of the List interface to reduce the amount of work required for classes that want to implement List.  
\* ***So basically if you want to implement the List interface and you don’t want to start from scratch and have to implement every single method, instead of just implementing List, you can extend this AbstractSequentialList class if you want a sequential list and you get a bunch of stuff for free and then you can just override what you want to override***.  
\* **But how is the LinkedList being stored?**  
=> **It’s similar to the way that we implemented our simple SinglyLinkedList and DoublyLinkedList, the LinkedList itself is the data structure. It’s not being backed by anything**.   
\* The LinkedList is just containing references to a head and a tail.  
\* The Node class:  
  
\* This should look familiar, and that’s why we went through those simple implementations, so you have some idea of what’s going on underneath the covers when you’re working with a LinkedList.  
**LinkedList remove()** => **removes the first item on the list**.  
**LinkedList removeFirst()  
LinkedList removeLast()**  
\* There are other methods, for example you could remvoe a specific item, of course that would mean the method has to search the list for that item and so it’s going to be a slower operation.  
\* There are also methods that let you insert an item at a specific point in the list - something like: insert item as the 6th item in the list. But again, those operations are going to be slower.  
\* The quickest operations are going to be those that are working at the head or at the tail.  
\* **The important point here is that if you want to use a LinkedList in Java, you can use the LinkedList class as long as you don’t mind the extra memory overhead due to the next and previous fields**.  
\* If you’re going to need a lot of nodes and memory is tight, you might want to consider another type of data structure, or if it makes sense, you might want to implement a SinglyLinkedList.  
\* **But most of the time, memory won’t be an issue**.  
\* There’s another type of Linked List called   
**CircularLinkedList** => this is a variation on the SinglyLinkedList and in this variation, the last node in the list doesn’t point to null. Instead, it loops back and it points to the head of the list and one advantage to this is that you can traverse the entire list, starting at any node.  
\* So if for some reason that feature is important to your application, then a CircularLinkedList might work for you.

**Linked Lists Challenge #1**  


**Linked Lists Challenge #1 Solution**  
**(Implemented the challenge before watching this video)**  
  
**Linked Lists Challenge #2**  
  
\* I want it to be a SinglyLinkedList, **this would be an easier challenge if it was a DoublyLinkedList**.

**Linked Lists Challenge #2 Solution**  
**(Implemented the challenge before watching this video)**  


**Resources**  
List interface Javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/List.html>  
ArrayList Class Javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/ArrayList.html>  
Vector Class Javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/Vector.html>  
LinkedList Class Javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/LinkedList.html>