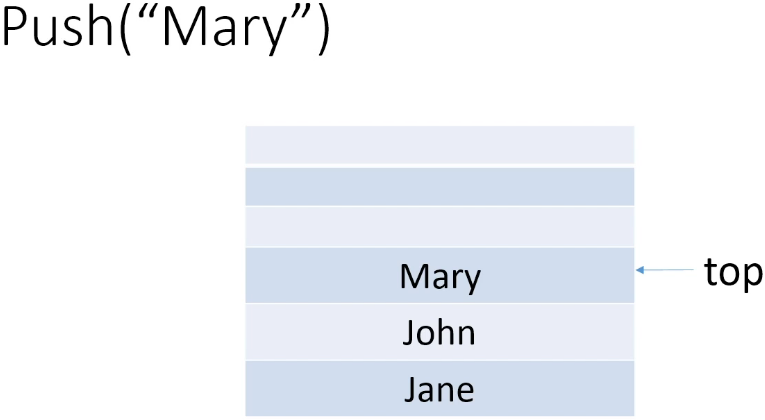
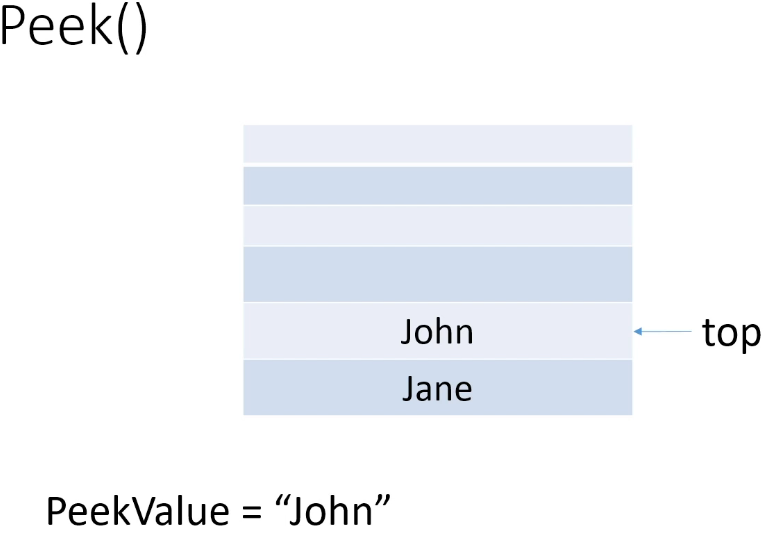
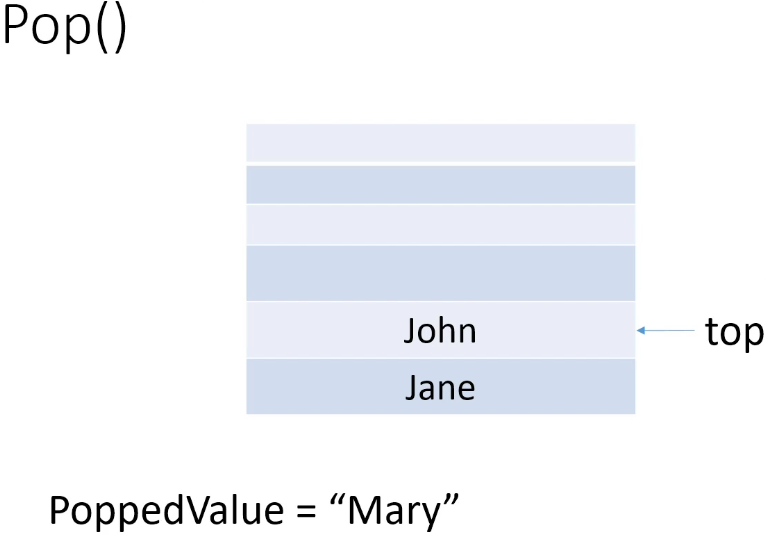
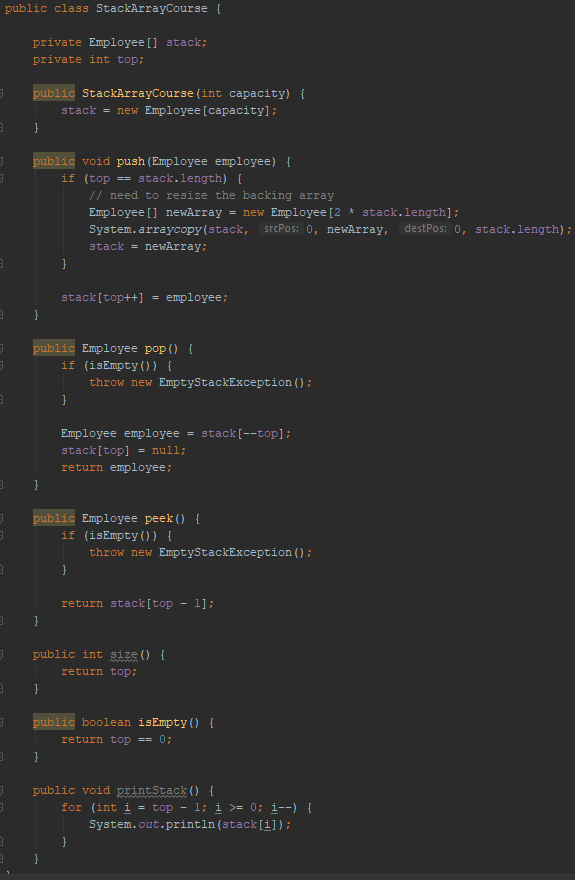
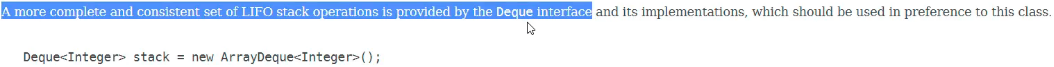
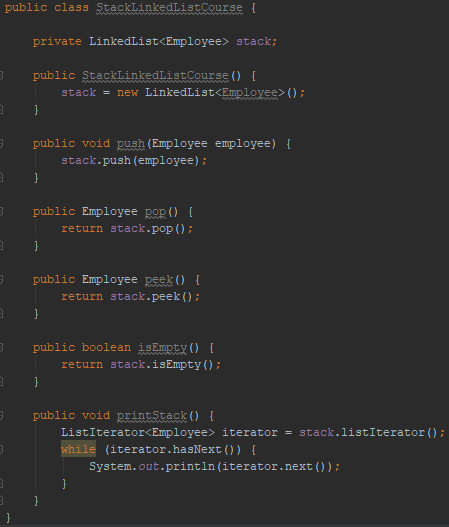
**Introduction to Stacks**  
\* **Instead of dictating how we store items, stacks dictate what operations we can do on the item**.  
\* Stack is known as an **Abstract Data Type**.

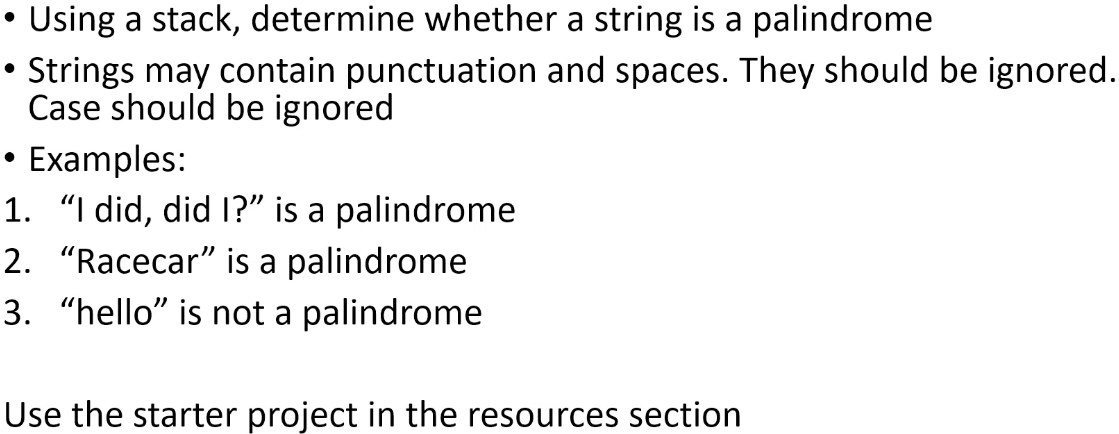
**Stacks (Theory)**  
\* **Conceptual** / **Abstract Data Type => what it does is it dictates the operations we can perform on a set of data, but it doesn’t tell us how the data should be stored.**  
\* A stack says: **I don’t care how the data is organized but these are the opeartions you’re allowed to do on the data**.  
=> Basically, **a stack can be backed by any data structure**.  
\* It doesn’t make sense for it to be backed by some data structures but you could back a stack with any data structure you wanted.  
\* **LIFO** => **Last in, first out** - one feature of stacks is that **the last item that you add to a stack, is always the first item that you can remove from the stack**.  
(life-o pronunciation)  
=> **That means there’s no random access, we can’t retrieve an element at a specific position**. You’re only ever allowed to remove the last item that you added to the stack.  
=> Because of that, the stack has the concept of a   
**top** => **the item at the top of the stack is the only item that we’re allowed to access**  
\* **We can perform 3 operations on a stack**. **1) push()** => adds an item as the top item on the stack  
**2) pop()** => removes the top item on the stack  
**3) peek()** => gets the top item on the stack without popping it  
\* **Linked List is the ideal backing data structure**.  
\* We’ve seen 1 use for a stack - **Call Stack** - when we looked at recursion, I talked about **pushing method calls onto the Call Stack and removing them**. And in that case **the last method that we called was always the first one that got taken off the Call Stack**. If method foo() called method bar(), it wouldn’t make sense for method foo() to return before method bar() does because foo() has to wait until bar() finishes executing.  
\* So a stack is the perfect data structure to use for the Call Stack.  
\* A Linked List is the perfect data structure for a stack, because because in a Linked List we always want to be **working with the item at the front** of the list if it’s a **Singly Linked List**. And so we add items at the front of the list and we remove items at the front of the list and by doing that, it means that the last item we added will always be the first item that we remove, and so a Linked List is a perfect data structure to back the stack.  
\* **push() would be addToFront()**  
\* **pop() would be removeFromFront()**  
\* **peek() would just give us the head item without removing it from the list**  
\* **But you could also back a stack with an array**.  
   
**O(1)** for **push**, **pop**, and **peek**, when using a **Linked** **List**.  
**O(n)** for **push**, if you use an **array** because the array may have to be resized.  
\* **If you know the max number of items that will be on the stack, an array can be a good choice**.  
\* **If memory is tight, an array might be a good choice because with an array, you don’t have the overhead of having to have a next field**.  
\* But if memory isn’t tight and if you don’t know what the maximum number of items is going to be, then Linked List is ideal.  
\* And even if memory is tight, a Linked List would be ideal if you don’t expect to be adding a whole ton of items.  
\* **A Linked List really does lend itself well to the behavioral restrictions that a stack has**.

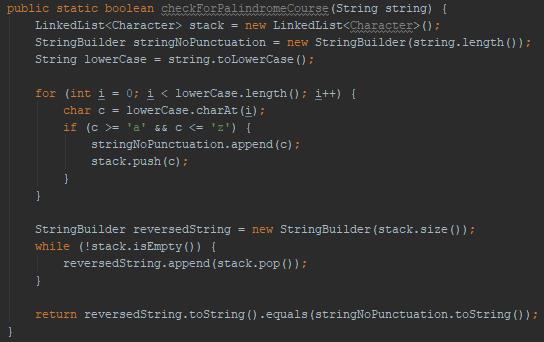
**Stacks Implementation (Array)**  
**(Implemented my own version of Stack using array and ArrayList)**  
\* Arrays are great for random access as we’ve learned. Do we need random access for a stack? No, we don’t, because we’re only ever going to be working with the top item on the stack.  
\* The other thing about arrays is of course they’re a fixed size. They’re not dynamic. And so by using an array to back a stack, we’re going to have to provide an initial size for the array and if we try to push something onto the stack when the array is full, then that’s obviously not going to work.  
\* For the same reason, using a ArrayList or a Vector wouldn’t be that great because they’re backed by an array.  
\* **Despite the problems of using an array, they are commonly used to back stack implementations**.  


\* The top is initialized to 0 by default.  
\* The top will always be the index where we would push the next element.  
\* In pop() if the stack is empty, we could return null, but instead we’re going to Throw an Exception.  
**throw new EmptyStackException**  
\* If we wanted to be really diligent, we could do a check in the pop() method and if we see there’s lots of empty space (by comparing the top value against the length of the array), we could consider resizing the array. **If we don’t worry about it, then the pop() operation will always be done in** **O(1) because it doesn’t depend on the number of items.**  
\* Resizing the array in the pop() method does come with a risk because if we then push more items we could end up having to resize the array again by making it larger.  
=> **Because of that, you’ll see implementations of stack that worry about resizing the array in the pop() method and you’ll see implementations that don’t**.  
\* **If you expect to push a ton of items onto the stack and then you’re going to be popping items off over time and never pushing items again, then resizing in the pop() method could make sense if memory is an issue**.  
\* **If memory isn’t an issue, it’s probably best to just leave the array at the size it is when popping items off**.  
\* **If you know the maximum number of data items you’re going to have upfront, then you’ll be able to set the appropriate capacity for the array and you won’t have to worry about the array being resized**.  
=> **That would mean all of your stack operations can be done in constant time, which would be fantastic**.  
=> **If you don’t expect your array to be resized frequently**, then using an array to implement a Stack is a good choice.  
=> **If you don’t know the maximum number of data items** that will ultimately be pushed onto the stack and so it’s possible the array may have to be resized frequently, or if you’re going to have to worry about resizing the array on a pop() because let’s say memory is really tight for some reason so you don’t want a lot of wasted space in the backing array then an array may not be a good choice, you might want to use a stack backed by a Linked List instead.

**Stacks Implementation (Linked List)**  
**(Implemented my own version of Stack using Singly Linked List)**  
<https://docs.oracle.com/javase/9/docs/api/java/util/Stack.html>  
\* Our first thought might be that JDK has a Stack class so we can just use it - but, if we read the documentation, we’ll see that the **Java team recommends that we not use this class**:  
  
\* Remember because Stack is an Abstract Data Type, the behavior is most likely going to be dictated by an Interrface and so it’s saying that instead of using the Stack class, **we should use a class that implements the Deque interface**.  
\* In the Queue section we’ll learn more about what this Deque interface is, but we can see in the docs that there is an **ArrayDeque** class.  
=> **That means it’s a Deque backed by an array**.  
=> **So if we wanted an array implementation of a stack, this is one of the classes we could use**.  
\* But we’ve already seen another class that implements the Deque interface and that’s the   
**LinkedList** class in the JDK and implementing a stack using a LinkedList is a great choice because we don’t have to worry about resizing the stack when we’re pushing items.  
=> If you want a stack in your Java application, you can go ahead and just use the LinkedList class.  
\* One potential problem with this is the class **allows us to do more than push(), pop() and peek()** items. **You could limit yourself in your code to those methods**.  
<https://docs.oracle.com/javase/9/docs/api/java/util/Deque.html>  
<https://docs.oracle.com/javase/9/docs/api/java/util/LinkedList.html>  
\* Because the Deque has push(), pop(), peek() methods and LinkedList implements it, it means that LinkedList has to have them as well. But it has a lot of other methods.  
\* So you have 2 choices:  
1) => Enforce that yourself in your code to use only those 3 methods.  
2) => Create a class that has this LinkedList as a field. So essentially we create a Stack class and it would have a field for the LinkedList and so the only methods you would expose from that class would be push(), pop() and peek().  
\* So instead of using our own implementation of a LinkedList and using that to back a stack, we’re going to use the JDK LinkedList class and we’re going to write an implementation that uses this class to back a stack.  
  
\* **Remember that this LinkedList is actually a Doubly Linked List** and for a stack you don’t need a Double Linked List because you’re always working with the item that’s at the front of the list.  
\* **But normally memory won’t be an issue** and so you can just go ahead and use this LinkedList class.  
\* If memory was an issue, then you would have to write your own Singly Linked List class or use an array instead. You could look at the other classes that implement the Deque interface and see if one of those would be more appropriate.  
**ListIterator<Employee> x = LinkedList listIterator()  
=> Traverses the list from head to tail and in a LinkedList implementation of a stack, the top item is always at the head.  
ListIterator hasNext()  
ListIterator next()**  
\* We went through this exercise because we want users of our stack that’s backed by a LinkedList to only be able to call push(), pop() and peek().  
\* If we used the LinkedList directly as a stack, then somebody might come along, perhaps a new developer on the team or something like that who is not familiar with what we’re doing and start calling other methods and we don’t want that to happen. We want a class that behaves like a stack.

\* push() => adding it at the head of the list.  
\* pop() => removing it from the head of the list.  
\* peek() => returning the head of the list.  
**O(1)** => with a LinkedList implementation, the Time Complexity for push(), pop() and peek().  
\* That’s because we’re always working with the item at the head of the list and so the steps involved don’t depend on the number of items in the list.  
\* **All a stack does is limit the operations you can do on whatever data structure is being used to implement the stack**.

**Stacks Challenge**  
  
\* You can code it however you want, as long as your solution uses a stack.  
\* You can use any class you want as the stack.  
\* As long as you’re using the LIFO behavior of stacks.

**Stacks Challenge Solution**  
**(Implemented the challenge)**  
\* **Stack is ideal for checking for palindromes** because what we basically want to do is reverse the string and then compare it against the original string and if they’re equal, then it’s a palindrome.  
\* Because it’s LIFO, if we push the string’s characters and then pop them, they will be reversed.  
\* The wrinkle here is the punctuation and the spaces.   
=> The approach I took was using the LinkedList class for the stack and I have a   
**StringBuilder** that I use to build up the original string with punctuation and spaces stripped out and the string is also lower cased. And as I’m doing that, I’m pushing characters that aren’t punctuation and aren’t spaces onto the stack.  
  
\* Just like with algorithms having many implementations, challenges can have many solutions.  
\* I could’ve taken the approach to loop through the string and popped out characters and compare them but instead, I’m just going to pop all of the characters off the stack and append them to another StringBuilder.

**Resources**  
Stack in JDK  
<https://docs.oracle.com/javase/9/docs/api/java/util/Stack.html>