# Workshop Create Linked List

## Overview.

In this lab we are going to implement our own **Linked List** data structure.  
A linked list should consists of **nodes**, also we'll implement **methods** to make the structure easy to use.



This figure shows how the structure should look. Specifically we are going to implement a **doubly linked list**. We have **nodes**, each node has **two reference properties** pointing to other nodes and **a value property** which contains some kind of data. By definition, the doubly linked list has a head (list start) and a tail (list end).

The typical operations over a doubly linked list are **add** / **remove** element at **both ends** and **traverse**.

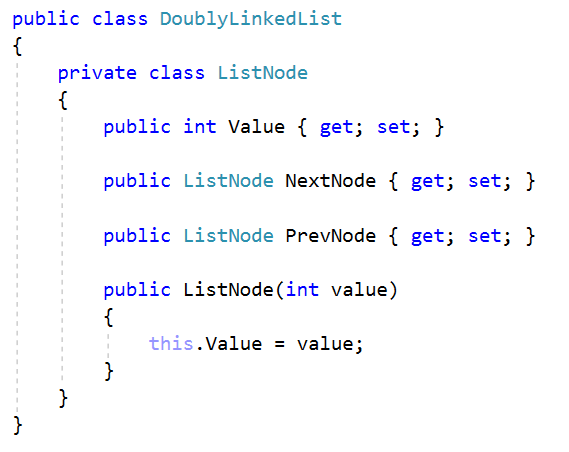
You can find more detailed information here: <https://en.wikipedia.org/wiki/Doubly_linked_list>

## Implement ListNode

The first step when implementing a linked / doubly linked list is to understand that we need **two classes**:

* ListNode class to hold a single list node (its value + next node + previous node)
* DoublyLinkedList to hold the entire list (its head + tail + operations)

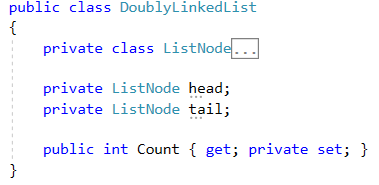
Now, let's write the **list node class**. It should hold a Value and a reference to its previous and next node. It can be inner class, because we will need it only internally from the doubly linked list class:



The class ListNode is called **recursive data structure**, because it references itself recursively. In this case our nodes' value property will be type of **int,** later in the course we will refactor the structure to be more generic.

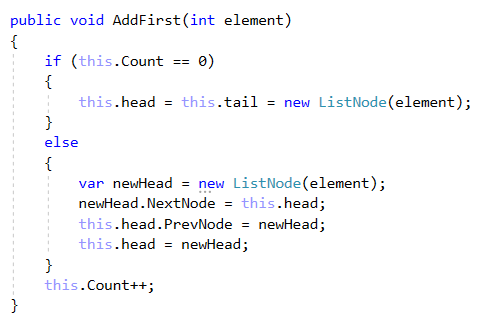
## Implement Head, Tail and Count

Now, let's define the head and tail of the doubly linked list:



## Implement AddFirst(int) Method

Next, implement the AddFirst(element) method:



Adding an element at the start of the list (before its head) has **two scenarios** (considered in the above code):

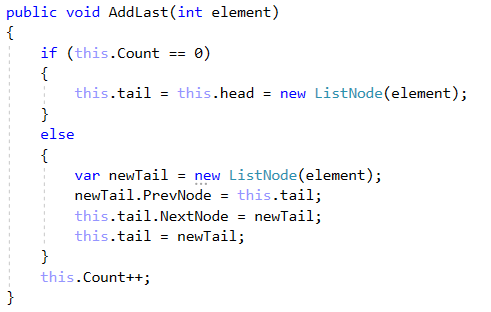
* **Empty list** 🡪 add the new element as head and tail in the same time.
* **Non-empty list** 🡪 add the new element as **new head** and redirect the **old head** as second element, just after the new head.



The above graphic visualizes the process of inserting a new node at the start (head) of the list. The **red** arrows denote the removed pointers from the old head. The **green** arrows denote the new pointers to the new head.

## Implement AddLast(int) Method

Next, implement the AddLast(int element) method for appending a new element as the list tail. It should be very similar to the AddFirst(int element) method. The logic inside it exactly the same, but we append the new element at the tail instead of at the head.

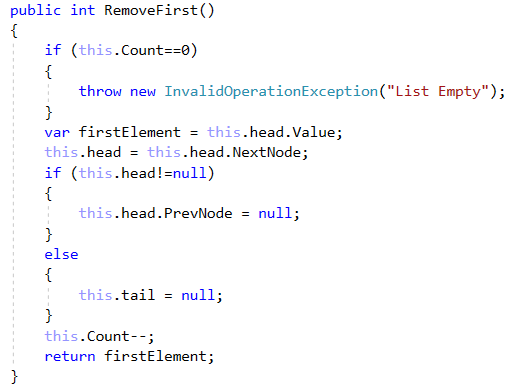


## Implement RemoveFirst() Method

Next, let's implement the method RemoveFirst() 🡪 int. It should **remove the first element** from the list and move its head to point to the second element. The removed element should be returned as a result from the method. In case of empty list, the method should throw an exception. We have to consider the following three cases:

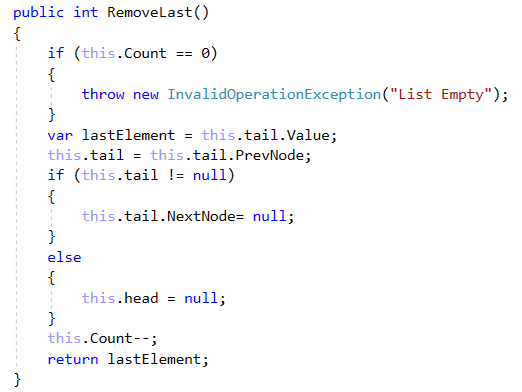
* **Empty list** 🡪 throw and exception.
* **Single element in the list** 🡪 make the list empty (head == tail == null).
* **Multiple elements in the list** 🡪 remove the first element and redirect the head to point to the second element (head = head.NextNode).

A sample implementation of RemoveFirst() method is given below:



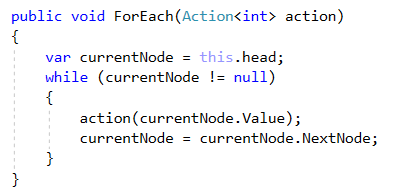
## Implement RemoveLast() Method

Next, let's implement the method RemoveLast() 🡪 int. It should **remove the last element** from the list and move its tail to point to the element before the last. It is very similar to the method RemoveFirst().



## Implement ForEach(Action) Method

We have a doubly linked list. We can add elements to it. But we cannot see what's inside, because the list still does not have a method to traverse its elements (pass through each of them, one by one). Now let's define the ForEach(Action<int>) method. In programming such a method is known as ["**visitor**" pattern](https://en.wikipedia.org/wiki/Visitor_pattern). It takes as an argument a function (action) to be invoked for each of the elements of the list. The algorithm behind this method is simple: start from head and pass to the next element until the last element is reached (its next element is null). A sample implementation is given below:



For example if you want to print all of the elements you can use the following code:

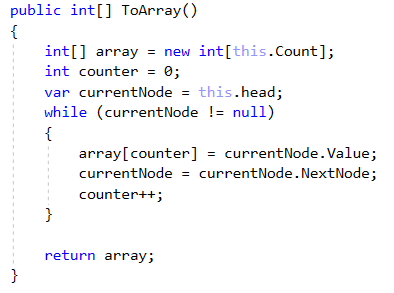


Where **list** is **DoublyLinkedList** type object.

## Implement ToArray() Method

Now, implement the next method: ToArray() 🡪 int[]. It should copy all elements of the linked list to an array of the same size. You could use the following steps to implement this method:

* Allocate an array int[] of size this.Count.
* Pass through all elements of the list and fill them to int[0], int[1], …, int[Count-1].
* Return the array as result.



Congratulations! You have implemented your doubly linked list.