# Exercises: Implement Doubly Linked List in C#

This document defines the **in-class exercises** assignments for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1147/Data-Structures-June-2015). You have to implement a **doubly linked list** in C# – a data structure that holds **nodes**, where each node knows its **next** and **previous** nodes:



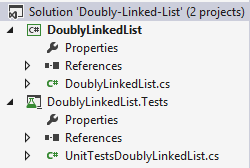
## Learn about Doubly Linked List in Wikipedia

Before starting, get familiar with the concept of doubly linked list: <https://en.wikipedia.org/wiki/Doubly_linked_list>.

The typical operations over a doubly linked list are **add** / **remove** element at **both ends** and **traverse**. By definition, the doubly linked list has a head (list start) and a tail (list end). Let's start coding!

## DoublyLinkedList<T> – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the DoublyLinkedList<T> class. The project holds the following assets:

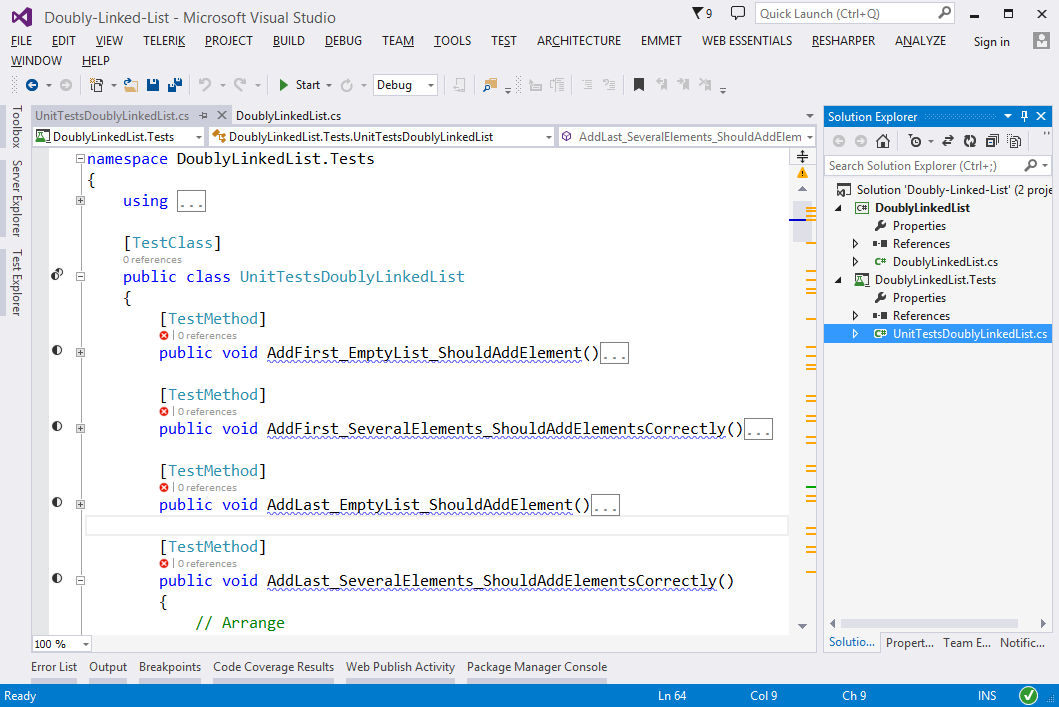


The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**.

The main class stays in the file DoublyLinkedList.cs:

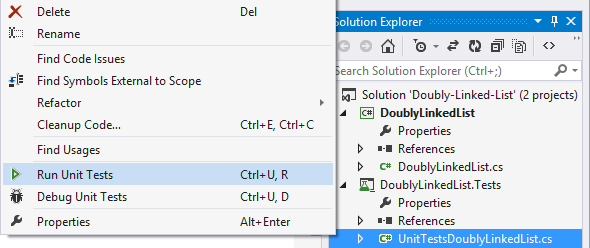
|  |
| --- |
| public class DoublyLinkedList<T> : IEnumerable<T>  {  public int Count { … }  public void AddFirst(T element) { … }  public void AddLast(T element) { … }  public T RemoveFirst() { … }  public T RemoveLast() { … }  public void ForEach(Action<T> action) { … }  public IEnumerator<T> GetEnumerator() { … }  IEnumerator IEnumerable.GetEnumerator() { … }  public T[] ToArray() { … }  } |

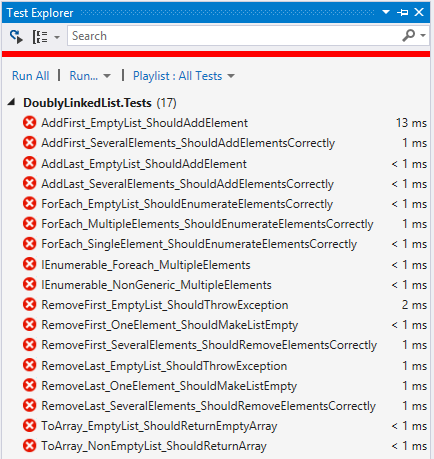
The project comes with **unit tests** covering the entire functionality of the doubly linked list (see the class UnitTestsDoublyLinkedList):



## Run the Unit Tests to Ensure All of Them Fail

**Run the unit tests** from the DoublyLinkedList.Tests project. Right click on the file "UnitTestsDoublyLinkedList.cs" in Solution Explorer and select **[Run Unit Tests]**:





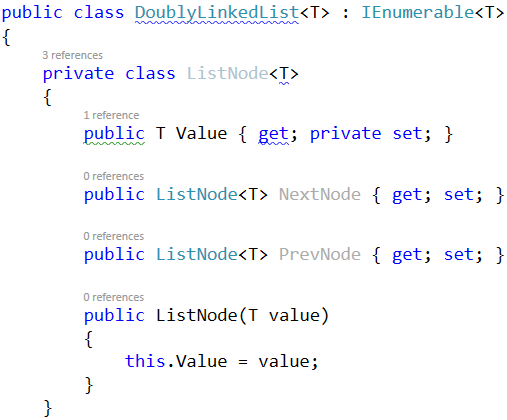
This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

## Implement ListNode<T>

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

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

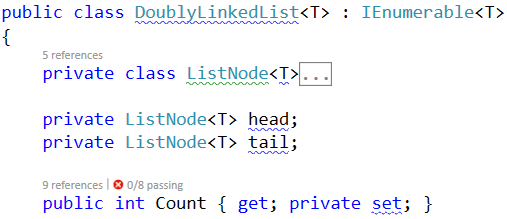
Now, let's write the **list node class**. It should hold a Value and a reference to its previous and next node. It can be inner class, because we will need it only internally from the doubly linked list class:



The class ListNode<T> is called **recursive data structure**, because it references itself recursively. It uses the **generic argument** T to avoid later specialization for any data type, e.g. int, string or DateTime. The **generic classes in C#** work similarly to **templates in C++** and **generic types in Java**.

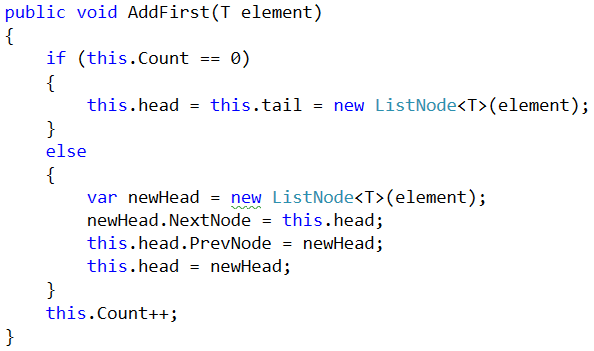
## Implement Head, Tail and Count

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



## Implement AddFirst(T) Method

Next, implement the AddFirst(T element) method:



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

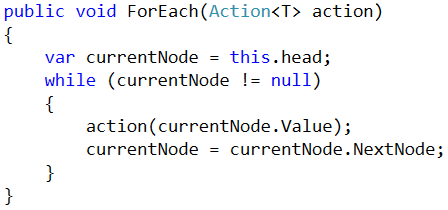
* **Empty list** 🡪 add the new element as head and tail in the same time.
* **Non-empty list** 🡪 add the new element as **new head** and redirect the **old head** as second element, just after the new head.



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.

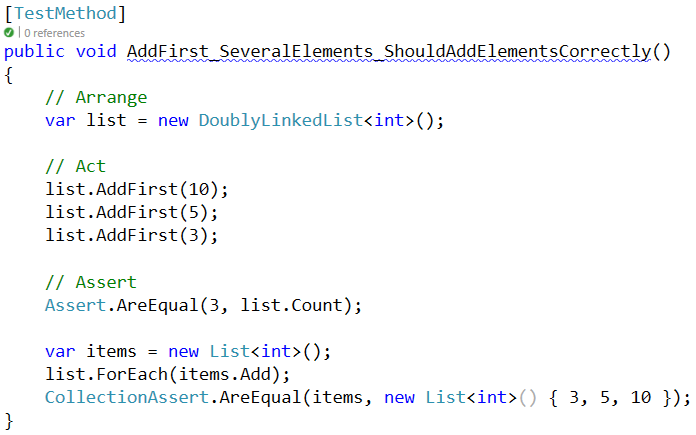
## Implement ForEach(Action) Method

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<T>) 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:

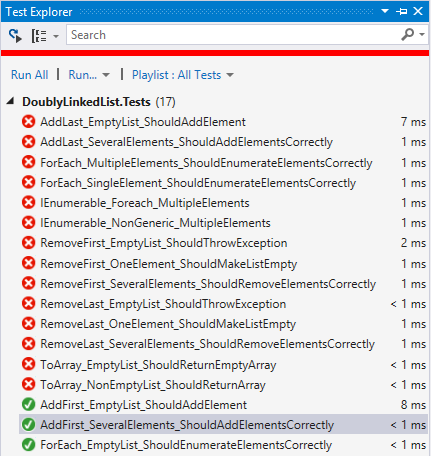


## Run the Unit Tests

Now we have the methods AddFirst(T) and ForEach(Action<T>). We are ready to run the unit tests to ensure they are correctly implemented. Most of the **unit tests** create a doubly linked list, add / remove elements from it and then check whether the elements in the list are as expected. For example, let's examine this unit test:

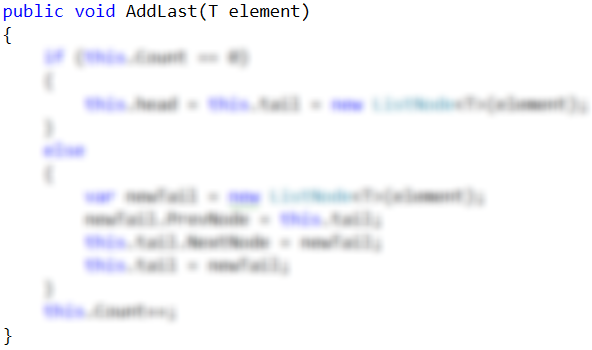


If we **run the unit tests**, some of them will now pass:

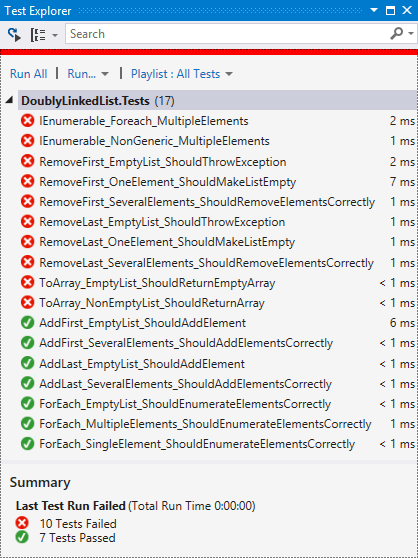


## Implement AddLast(T) Method

Next, implement the AddLast(T element) method for appending a new element as the list tail. It should be very similar to the AddFirst(T element) method. The logic inside it exactly the same, but we append the new element at the tail instead of at the head. The code below is intentionally blurred. Write it yourself!



Now **run the unit tests** again. You should have several more passed (green) tests:

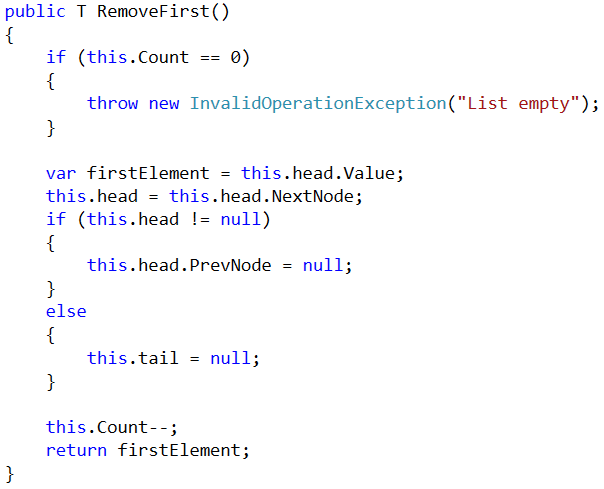


## Implement RemoveFirst() Method

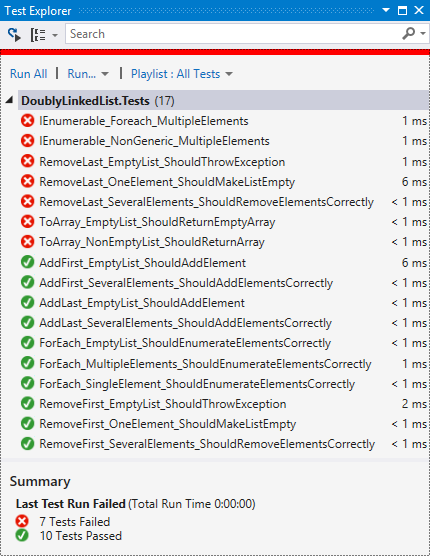
Next, let's implement the method RemoveFirst() 🡪 T. It should **remove the first element** from the list and move its head to point to the second element. The removed element should be returned as a result from the method. In case of empty list, the method should throw an exception. We have to consider 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:



Run the **unit tests** to ensure the method is correctly implemented:

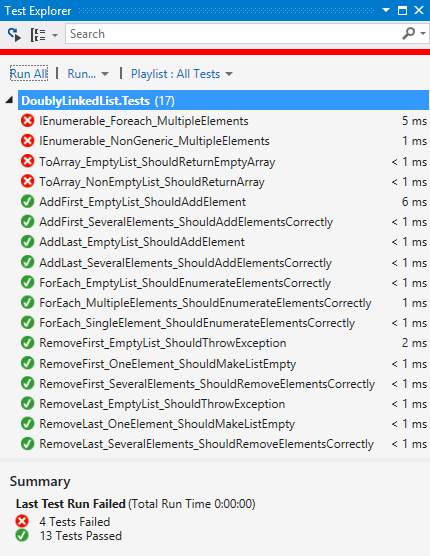


## Implement RemoveLast() Method

Next, let's implement the method RemoveLast() 🡪 T. 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(), so you are free to implement it yourself. The code below is intentionally blurred:



Now **run the unit tests** once again to ensure your code is correct:

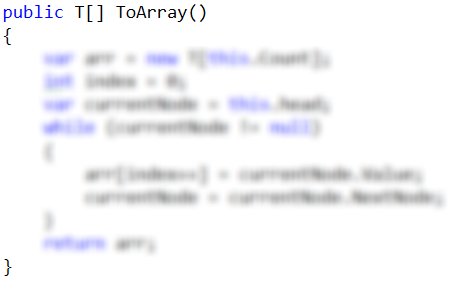


## Implement ToArray() Method

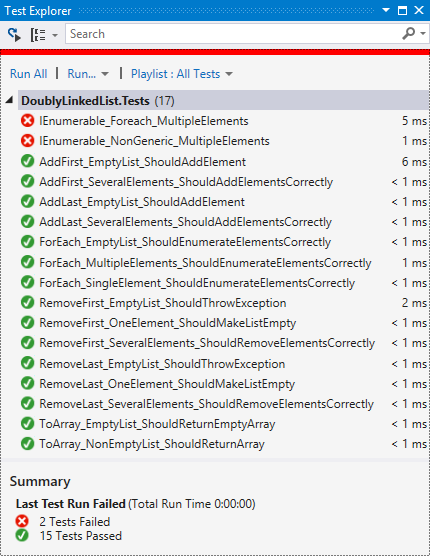
Now, implement the next method: ToArray() 🡪 T[]. It should copy all elements of the linked list to an array of the same size. You could use the following steps to implement this method:

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

Write yourself the blurred code in the method ToArray():

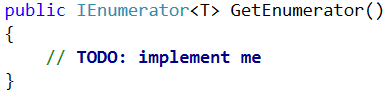


Again, **run the unit tests**, to ensure your code is correct:



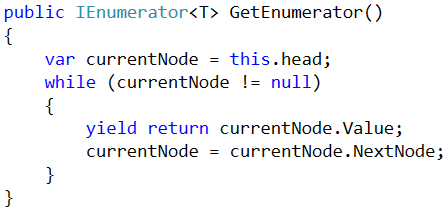
## Implement IEnumerable<T>

Collection classes in C# and .NET Framework (like arrays, lists and sets) implement the system interface IEnumerable<T> to enable the foreach iteration over their elements. The C# keyword foreach calls internally the following method:



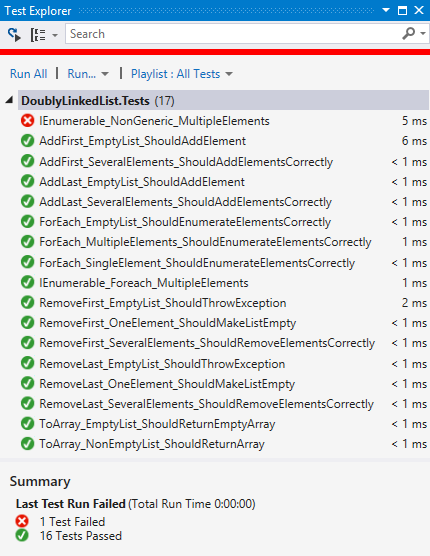
This method returns IEnumerator<T>, which can move to the next element and read the current element. In programming, this is known [as "**iterator**" pattern](https://en.wikipedia.org/wiki/Iterator_pattern) (**enumerator**).

We will use [the "**yield return**" C# statement](https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx) to simplify the implementation of the iterator:

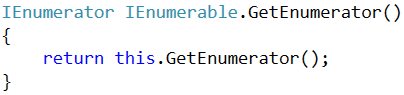


The above code will enable using the DoublyLinkedList<T> in foreach loops.

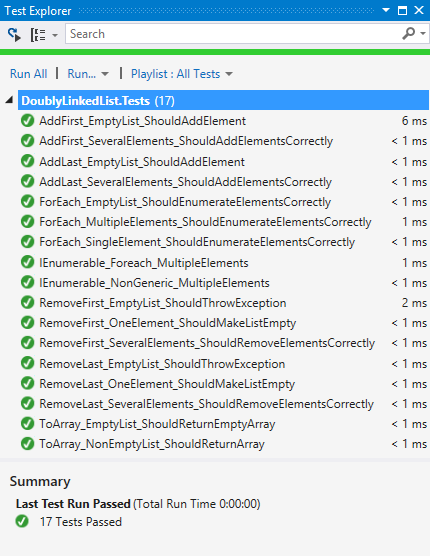
Now, we have added the iterator over the list elements, so let's **run the unit tests** again:



We have all but one unit tests passed. The last unimplemented method is the **non-generic enumerator**:



Finally, **run the unit tests** to ensure all of them pass correctly:



Congratulations! You have implemented your doubly linked list.