1. LINQ Append Method in C#:

The Append method in LINQ adds a single element to the end of an IEnumerable<T> sequence. This method is available in .NET Core and .NET Standard 2.0 and later, as well as in the .NET Framework starting with version 4.7.1.

The LINQ Append Method in C# is used to append a value to the end of the sequence. This method does not modify the elements of the sequence. Instead, it creates a copy of the sequence with the new appended element. If you go to the definition of the Append Method, then you will see the following signature.



**Type Parameters**

TSource: The data type of the elements contained in the sequence.

**Parameters:**

IEnumerable<TSource> source: A sequence of values.

TSource element: The value to append at the end of the sequence.

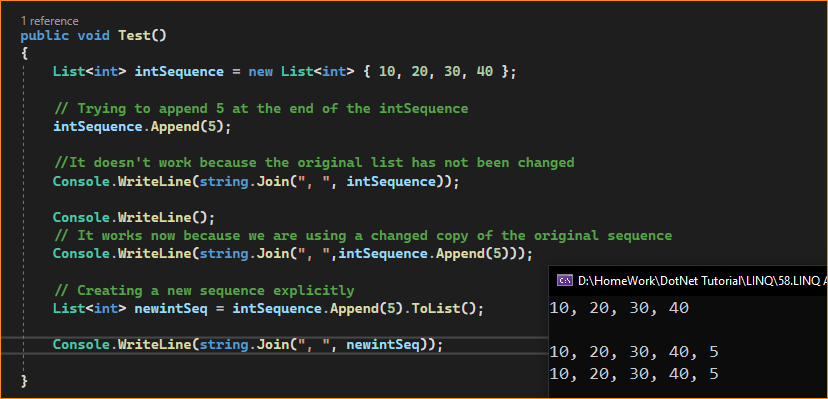
**Returns:**

IEnumerable<TSource>: A new sequence that ends with the element.

Exceptions: When the source is null, it will throw ArgumentNullException.

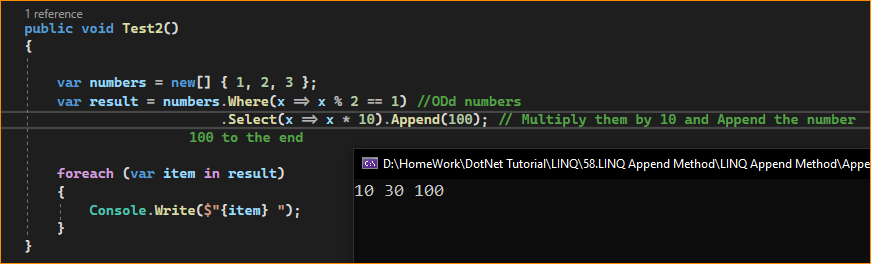
1. Example to Understand LINQ Append Method in C#:

The following example shows how to use the Append Method to append a value to the end of the sequence. The following example is self-explained. So, please go through the comment lines.



The Append method is useful when you have an existing sequence and want to add an element to the sequence without modifying the original sequence. It is important to note that, like many other LINQ methods, Append uses deferred execution. The appended element isn’t added to the sequence until you iterate.

This method is also commonly used in a method chain, where you might filter or project elements and then append new ones. Here’s a slightly more complex example involving method chaining:



Note: When using Append, remember that each call will create a new IEnumerable<T> instance, which can have performance implications if used excessively within a loop or on large sequences. It’s also not suitable for adding multiple elements; for that, you would typically use the Concat method.

1. When to Use the LINQ Append Method in C#?

The Append method in LINQ is relatively new, and it provides a convenient way to add a single item to the end of an IEnumerable<T> sequence without modifying the original sequence. Here are some scenarios where it might be appropriate to use the Append method:

**1. Adding a Single Element**

Use Case: Creating a new collection or using Concat for a single item would be overkill when you need to add just one element to a sequence.

Example: You have an array of user IDs and want to add a new user ID to the end.

**2. Fluent Chaining**

Use Case: When using method chaining to perform a series of LINQ operations and want to add an element at the end fluently.

Example: You’re filtering a list of dates and want to add the current date to the end of the sequence.

**3. Streamlined Syntax**

Use Case: When you prefer a concise syntax that doesn’t require the creation of a new array or list to add a single item.

Example: If certain conditions are met, you have a sequence of status codes and want to add a default status code to the sequence.

**4. Immutability**

Use Case: When working with immutable sequences, you cannot or do not want to modify the original sequence.

Example: You have an immutable list of configuration settings and want to add an additional setting before passing it to a consumer.

**5. Functional Programming Patterns**

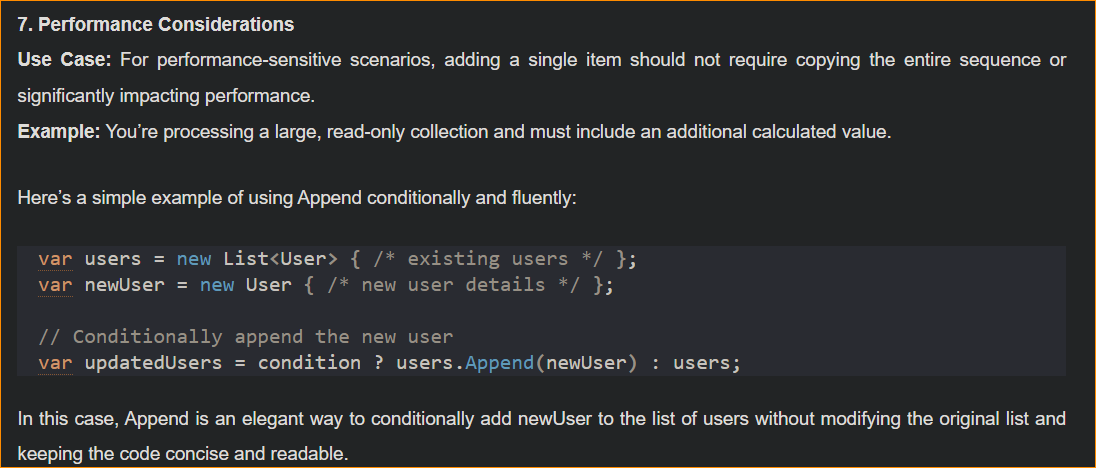
Use Case: When employing functional programming patterns that avoid mutating state.

Example: In a sequence of operations, you append an element to pass a complete sequence to the next function in the chain.

**6. Conditional Appending**

Use Case: When you want to append an element based on a condition without disrupting the flow of LINQ operations.

Example: Adding a ‘total’ row at the end of a sequence of rows based on a condition.



Remember that each call to Append creates a new IEnumerable<T>, and because LINQ uses deferred execution, the appending doesn’t happen until you enumerate over the sequence. Appending multiple items individually in a loop could lead to a performance hit due to the creation of multiple intermediate sequences. In such cases, it may be more efficient to use other collection manipulation methods, like AddRange on a List<T>.