List<T>

Topics Covered

- Generics and Type Placeholders
- Lists as Dynamic Arrays
- Constructors, Count, and Capacity
- Add, Remove and Insert Methods
- Search Methods
- Sorting Lists
- Binary Search
- Read-only Access
- Other Methods

List<T>

The **T** in List<T> is a type placeholder. We can substitute for T <u>any</u> non-static C# type.

We can declare (for example):

- List<int>
- List<string>
- List<double>
- List<DateTime>
- List<StringBuilder>
- List<FileStream>
- List<MathOperation>

Classes (and methods) that use a type placeholder inside angle-brackets are called generic classes (and methods).

Lists as Dynamic Arrays

List<T> is a dynamic array. It will grow in size as we add elements using the Add method.

The List<T> class defines an indexer so that we can access elements using array syntax.

List<T> provides methods that allow us to search, sort and modify the list.

List<T> implements IEnumerable, allowing it to be used in foreach loops.

List<T> is defined in the System.Collections.Generic namespace. Here is the class declaration:

```
[Serializable]
public class List<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>,
System.Collections.Generic.IList<T>,
System.Collections.Generic.IReadOnlyCollection<T>,
System.Collections.Generic.IReadOnlyList<T>, System.Collections.IList
```

List<T> Constructors

List<T> has 3 constructors.

The default constructor initializes a list with default capacity:

```
List<T>()
```

Another constructor initializes a list with a pre-set initial capacity:

```
List<T>(int capacity)
```

The third constructor initializes a list with an enumeration of elements:

```
List<T>(IEnumerable<T> items)
```

List Properties

Property	Description
public int Capacity { get; set; }	Gets or sets the total number of elements the internal data structure can hold without resizing.
public int Count { get; }	Gets the number of elements contained in the List<T>.
public T this[int index] { get; set; }	Gets or sets the element at the specified index.

List<T>.Add and List<T>.AddRange

Add adds a single item to the end of the list:

```
public void Add (T item);
```

The item is added and Count increases by 1.

Depending on the capacity, the method may trigger a re-allocation of the underlying storage.

AddRange adds multiple items to the list:

```
public void AddRange (IEnumerable<T> collection);
```

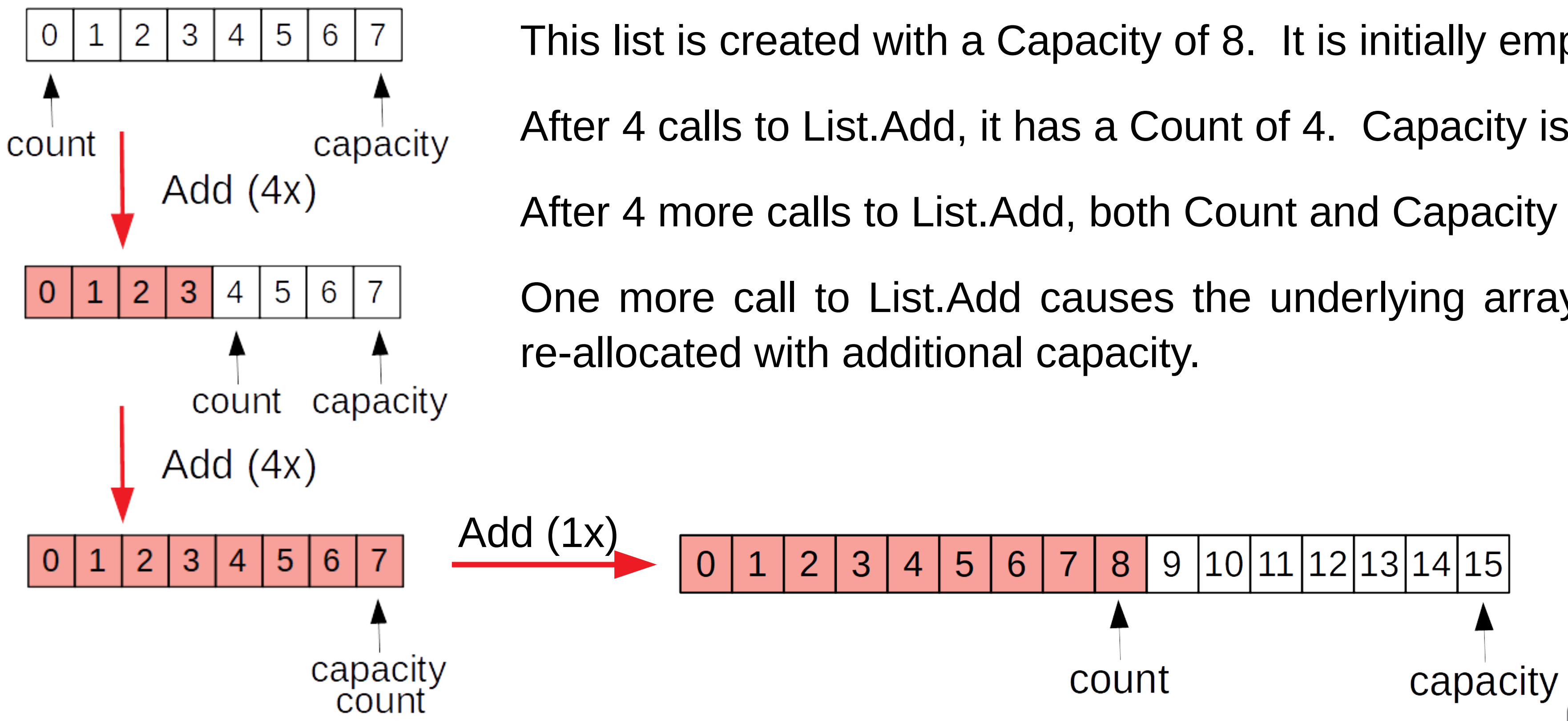
Count and Capacity

This list is created with a Capacity of 8. It is initially empty.

After 4 calls to List.Add, it has a Count of 4. Capacity is still 8.

After 4 more calls to List.Add, both Count and Capacity are 8.

One more call to List.Add causes the underlying array to be re-allocated with additional capacity.



Remove Methods

The **Remove** method removes the first occurrence of a specific object from the List<T>:

```
public bool Remove (T item);
```

RemoveAt removes the element at the specified index of the List<T>:

```
public void RemoveAt (int index);
```

RemoveRange removes a range of elements from the List<T>:

```
public void RemoveRange (int index, int count);
```

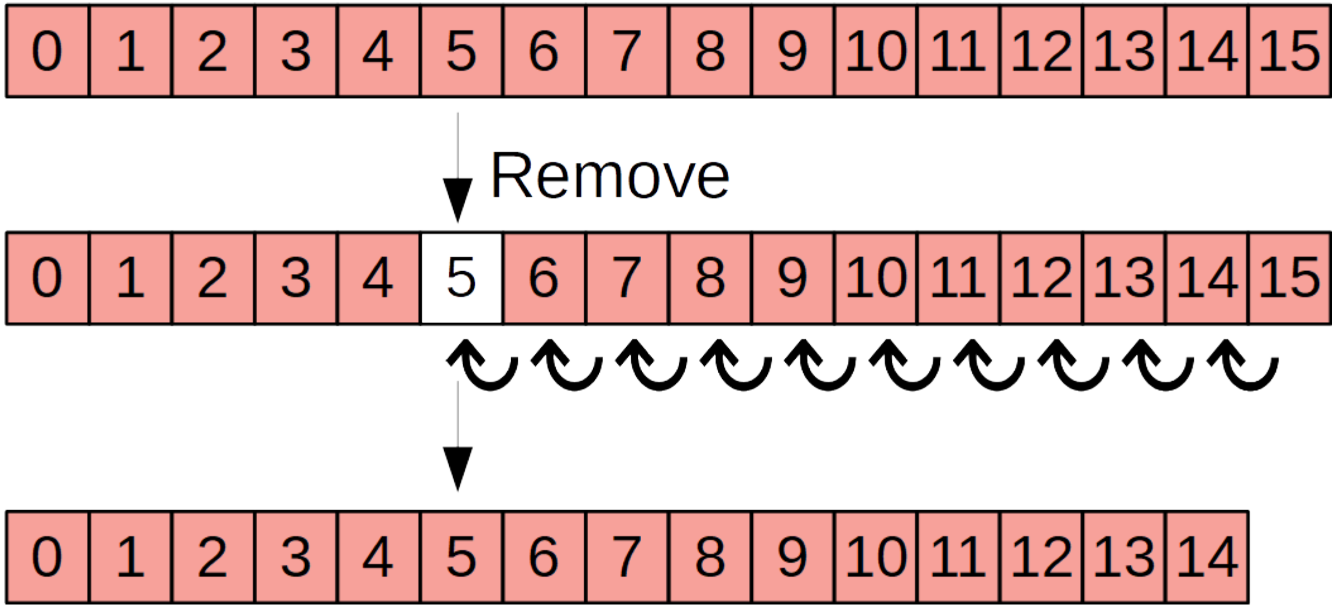
RemoveAll removes all the elements that match the conditions defined by the specified predicate:

```
public int RemoveAll (Predicate<T> match);
```

Clear removes all elements from the List<T>:

```
public void Clear ();
```

The Remove Operation



Insert Methods

Insert inserts an element into the List<T> at the specified index:

```
public void Insert (int index, T item);
```

InsertRange inserts the elements of a collection into the List<T> at the specified index:

```
public void InsertRange (int index, IEnumerable<T> collection);
```

Inserting an element into the list requires the same set of operations as Remove, but in reverse:

1. Reallocate the array if necessary;
2. Move all elements above the insertion point to the right;
3. Place the new element into the insertion point.

Search Methods

Contains determines whether an element is in the List<T>:

```
public bool Contains (T item);
```

Exists determines whether the List<T> contains elements that match the conditions defined by the specified predicate:

```
public bool Exists (Predicate<T> match);
```

Find searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>:

```
public T Find (Predicate<T> match);
```

FindAll retrieves all the elements that match the conditions defined by the specified predicate:

```
public List<T> FindAll (Predicate<T> match);
```

More Search Methods

FindIndex searches for an element that matches the conditions defined by a specified predicate, and returns the zero-based index of the first occurrence within the List<T>:

```
public int FindIndex (Predicate<T> match);
```

FindLast searches for an element that matches the conditions defined by the specified predicate, and returns the last occurrence within the entire List<T>:

```
public T FindLast (Predicate<T> match);
```

FindLastIndex searches for an element that matches the conditions defined by a specified predicate, and returns the zero-based index of the last occurrence within the List<T> or a portion of it:

```
public int FindLastIndex (Predicate<T> match);
```

Sorting Lists

List<T> has four overloads of its Sort method:

```
public void Sort ();
public void Sort (IComparer<T> comparer);
public void Sort (Comparison<T> comparison);
public void Sort (int index, int count, IComparer<T> comparer);
```

The first method sorts the elements in the entire List<T> using the default comparer.

The second and third methods allow us to provide in custom comparison logic.

The fourth method allows us to sort just a sub-range of the data using custom comparison logic.

BinarySearch

The search methods described over the past several slides use a sequential search – each element in the list must be tested until one element (or more) passes the search criterion. In a worst-case scenario, we must test every element in the list.

This is called O(N) performance, where N represents the # elements in the list.

If we know that the list is sorted, we can search long lists much more quickly using a binary search algorithm. Binary search runs with O(log N) performance.

List<T>.BinarySearch returns the index of an item in the list.

If the item is not in the list, returns the complement of the index at which the item should be inserted to preserve sorting.

Since indexes are always >=0, the complement is easy to recognize because it is a negative number.

The next slide shows how to use this to populate a list with unique values while maintaining sorting.

Using BinarySearch

```
Random rand = new Random();
List<int> values = new List<int>(100);
for(int i=0;i<100;++i)
{
    int next = rand.Next(0, 1000);
    int ndx = values.BinarySearch(next);
    if (ndx < 0) values.Insert(~ndx, next);
}
Console.WriteLine($"{values.Count} values inserted:");
Console.WriteLine(string.Join(" ", values));
```

ToArray

The List<T> generic class is designed to act as a dynamically-resizing array.

It then makes sense that the class provides a method that converts it to an actual array.

ToArray copies the elements of the List<T> to a <u>new</u> array.

Note that it is a <u>new</u> array. Changes to the array (such as setting an element to 0 or null) <u>will not change</u> the List from which it was generated.

```
public T[] ToArray ();
```

Read-Only Access to a List<T>

Numerous methods of the List<T> class modify the list – Add, Remove, Insert, Clear, etc.

Occasionally, a class maintains a List of objects and it needs to expose the List as read-only, allowing enumeration, inspection and searching, but not allowing modification.

With the List<T> class, we have two options.

- List<T> has a method which returns a ReadOnlyCollection<T>:

```
public ReadOnlyCollection<T> AsReadOnly ();
```

- We can also cast a list to any of the interfaces which it implements. IEnumerable<T> is particularly useful, and is read-only.

Other Methods

ConvertAll converts the elements in the current List<T> to another type, and returns a list containing the converted elements.:

```
public List<TOutput> ConvertAll<TOutput>(Converter<T,TOutput> converter);
```

ForEach performs the specified action on each element of the List<T>:

```
public void ForEach (Action<T> action);
```

Reverse reverses the order of the elements in the entire List<T>:

```
public void Reverse ();
```

Topics Covered

- Generics and Type Placeholders
- Lists as Dynamic Arrays
- Constructors, Count, and Capacity
- Add, Remove and Insert Methods
- Search Methods
- Sorting Lists
- Binary Search
- Read-only Access
- Other Methods

Questions

- What is the **T** in List<T>?
- What is the System.Collections.Generic namespace?
- What is the Capacity of a list. What is the Count of a list?
- List<T> defines an indexer. What can we do with the indexer?
- What happens when we remove an item from a list?
- What happens when we insert an item into a list?
- List<T>.Contains tests whether an item is in the list. How does it know if two items are equal?
- What happens when we sort a list?
- How does List<T>.BinarySearch differ from List<T>.Find?