Lesson 34 - Collections IV

`Stack<T>`

A `stack` is a **Last In, First Out (LIFO)** collection of elements where the last element that goes into the stack will be the first element that comes out.

Inserting an element onto a stack is called **pushing.** Deleting an element from a stack is called **popping**. Pushing and popping can be performed only at the top of the stack.

> Stacks can be used to create undo-redo functionalities, parsing expressions (infix to postfix/prefix conversion), and much more.

The C# generic collection `Stack<T>` class requires all elements to be of the same type `T`.

`Stack<T>` properties include:
`Count` - Returns the number of elements in the stack.

`Stack<T>` methods include:
`Peek()` - Returns the element at the top of the stack without removing it.
`Pop()` - Returns the element at the top of the stack and removes it from the stack.
`Push(T t)` - Inserts an element t at the top of the stack.

```csharp
Stack<int> st = new();

st.Push(60);
st.Push(70);
st.Push(10);

StackPrinter(st);

Console.WriteLine($"Peek: {st.Peek()}");
StackPrinter(st);

Console.WriteLine($"Count: {st.Pop()}");
StackPrinter(st);

static void StackPrinter(Stack<int> stack, string name="Stack")
{
    Console.WriteLine($"\n{name}: ");
    foreach (var item in stack)
        Console.Write(item + " ");
    Console.WriteLine($"Count: {stack.Count}");
}
```

Here are additional `Stack<T>` methods:
`Clear()` - Removes all the elements from the stack.
`Contains(T t)` - Returns true when the element t is present in the stack.
`ToArray()` - Copies the stack into a new array.

`List<T>`

A `list` is similar to an array, but the elements in a list can be inserted and removed **dynamically**. The C# generic collection `List<T>` class requires all elements be of the same type `T`.

`List<T>` properties and methods include:

- `Count` A property that gets the number of elements contained in the list.
- `Item[int i]` Gets or sets the element in the list at the index i. Item is the indexer and is not required when accessing an element. You only need to use the brackets [] and the index value inside the brackets.
- `Add(T t)` Adds an element t to the end of the list.
- `RemoveAt(int index)` Removes the element at the specified position (index) from the list.
- `Sort()` Sorts elements in the list.

```
List<int> li = new();

li.Add(60);
li.Add(70);
li.Add(10);
li.RemoveAt(1);

ListPrinter(li);

li.Sort();
ListPrinter(li, "Sorted");

static void ListPrinter(List<int> list, string name="List")
{
    Console.WriteLine($"\n{name}: ");
    foreach (var item in list)
        Console.Write(item + " ");
}
```

Additional `List<T>` properties and methods are listed below. Try them out by adding them to the `List<T>` example code above.

`Capacity` - A property that gets the number of elements the list can hold before needing to be resized.

`Clear()` - Removes all the elements from the list.

`TrimExcess()` - Sets the capacity to the actual number of elements in the list. This is useful when trying to reduce memory overhead.

`AddRange(IEnumerable coll)` - Adds the elements of collection coll with elements of the same type as `List<T>` to the end of the list. IEnumerable is the collections interface that supports simple iteration over the collection.

`Insert(int i, T t)` - Inserts an element t at a specific index i in the list.

`InsertRange(int i, IEnumerable coll)` - Inserts the elements of a collection coll at a specified index i in the list. IEnumerable is the collections interface that supports simple iteration over the collection.

`Remove(T t)` - Removes the first occurrence of the object t from the list.

`RemoveRange(int i, int count)` - Removes a specified number of elements count from the list starting at a specified index i.

`Contains(T t)` - Returns true if the specified element t is present in the list.

`IndexOf(T t)` - Returns the index of the first occurrence of the element t in the list.

`Reverse()` - Reverses the order of the elements in the list.

`ToArray()` - Copies the elements of the list into a new array.