Arrays and Lists

Tracking more than one

So far we've only been able to keep track of one thing per variable.

Keeping track of more than one thing.

```
var name1 = "Mark";
var name2 = "Paula";
var name3 = "Sandy";
var name4 = "Bill";
```

Arrays

```
//                 Type
//                 |
//                 |   Array
//                 |    ||
//                 |    ||  Start of list of values
//                 |    ||   |
//                 |    ||   |           Values
//                 |    ||   |            |
//                 |    ||   |            |                        End of list
//                 |    ||   |            |                         |
//                 |    ||   |            |                         |
//                 v    vv   v            v                         v
var names = new string[]   { "Mark", "Paula", "Sandy" , "Bill" };
```

Accessing elements

Bring back our friend the `[]` index operator.

Similar to accessing the individual characters in a string, `[]` allows us to access the individual elements of an array.

Indexes start at `0`.

```
var firstName = names[0];
var secondName = names[1];
```

We can also ask for the length of the array.

```
var nameCount = names.Length;
```

Unfortunately `arrays` come with some limitations:

- Once an array is created, it's size cannot change
- If we access an index that does not exist our program will crash. For example `names[42]` will cause our program to have an exception and stop.
- Arrays can only store data of the same type

Welcome to: List

Luckily there is a more flexible type that provides the features we'd want from a list of data.
Even more fortunate, it is conveniently named: `List`

To create a list of strings with some data:

```
//
//                Making a new List
//                 |
//                 |     ... of strings
//                 |     |
//                 |     |           Start of initial list of strings
//                 |     |           |
//                 |     |           |       Values
//                 |     |           |       |
//                 |     |           |       |                    End of list
//                 |     |           |       |                              |
//                 |     |           |       |                              |
//                 v     |           v       v                              v
var names = new List<string>()  { "Mark", "Paula", "Sandy" , "Bill" };
```

Adding a `using` statement

If you add this code to a program you will notice red-squiggly errors for `List`.

The `List` is not an intrinsic type. This means that we need to tell `C#` to use it in our code.

To do this we need to add a `using` statement. `using` tells `C#` what other code ours depend on.

VS Code Auto fix

Luckily VS Code is very smart.

We can click on `List` and press **Control .** -- or click on the **lightbulb** icon to see a list of quick fixes.

We want to use the one that adds a `using` statement.

In this case `using System.Collection.Generic` -- This is the **namespace** where the `List` code lives.

Now that we have our list we can work with it.

The `List` is still accessed with the `[]` bracket syntax and is still zero indexed:

```
var firstName = names[0];
var secondName = names[1];
```

We can find out how many elements are in the list but we must use a new syntax:

```
var numberOfNamesInList = names.Count;
```

For `List` we use `Count` instead of `Length`

Since these `Lists` are more flexible we can **add** elements to the list.

```
names.Add("George");
```

We can create `Lists` of other types as well.

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);
```

Other helpful List features

Clear

This method removes all the elements from the current list.

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);

listOfScores.Clear();
```

IndexOf

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);

var indexOfFiftyFive = listOfScores.IndexOf(55);
Console.WriteLine($"Found 55 at index {indexOfFiftyFive}"); // Prints 2
```

Insert

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);

// Insert the number `42` so it placed at index 2 (in this case after the 100)
listOfScores.Insert(2, 42);

// Now our list has: 12, 100, 42, 55, and 44
```

Remove

Removes the value from the list. Only the first occurrence of the value is removed. For instance in the example below, there are two 55s in the list. After calling `Remove(55)` there is still one 55 in the list.

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);
listOfScores.Add(55);

Console.WriteLine($"Our list has {listOfScores.Count} values"); // Prints 5
listOfScores.Remove(55);

Console.WriteLine($"Our list has {listOfScores.Count} values"); // Prints 4
```

RemoveAt

If we want to remove a value at a specific instance we can call RemoveAt and supply the index.

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);
listOfScores.Add(55);

Console.WriteLine($"Our list has {listOfScores.Count} values"); // Prints 5

// This will remove the `44` since it's index 3.
listOfScores.RemoveAt(3);

Console.WriteLine($"Our list has {listOfScores.Count} values"); // Prints 4
```

Reverse

This reverses the list in place.

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);
listOfScores.Add(55);

listOfScores.Reverse();

// Now our list has 55 44 55 100 12
```

Sort

This orders the values in place.

```
var listOfScores = new List<int>();
listOfScores.Add(12);
listOfScores.Add(100);
listOfScores.Add(55);
listOfScores.Add(44);
listOfScores.Add(55);

listOfScores.Sort();

// Now our list has 12 44 55 55 100
```

We will be using `List` quite a lot in our time with `C#`.

Top Tip: See the Quick Reference Guide for more useful `List` help