28. What are indexers?

Brief summary: Indexers allow instances of a type to be indexed just like arrays. In this way, they resemble properties except that they take parameters. For example, a Dictionary<string, int> has an indexer that allows calling "dictionaryVariable["some key"]" to access the value under some key.

Indexers are something we use all the time, usually without giving it much thought. Whenever accessing an element under a specific index of a list, we are actually calling the List's indexer:

```
var list = new List<decimal> { 5.5m, 3m, 0m };
var thirdElement = list[2];
```

This is exactly the same as accessing the third element of an array. This code may seem simple, but there is a lot going on in the List class whenever we use its indexer. Let's take a look into List's source code:

```
// Sets or Gets the element at the given index.
//
public T this[int index] {
    get {
        // Following trick can reduce the range check by one
        if ((uint) index >= (uint)_size) {
            ThrowHelper.ThrowArgumentOutOfRangeException();
        }
        Contract.EndContractBlock();
        return _items[index];
    }

    set {
        if ((uint) index >= (uint)_size) {
            ThrowHelper.ThrowArgumentOutOfRangeException();
        }
        Contract.EndContractBlock();
        _items[index] = value;
        _version++;
    }
}
```

We can recognize the definition of the indexer by the "this[someType paramName]" code. For the case of a List, the indexer is quite complex, but in the end, its main job is to get or set the value of an internal array called "_items". In the lecture "What is a List?" we learned that they use an array as the underlying collection, and this is exactly what we see here.

Indexers don't necessarily use integers as parameters. For example, when using Dictionaries we use indexers to access an element under a given key - and a key of a Dictionary can be whatever we want. For example, if the key of the Dictionary is a string, then the parameter of the indexer will also be a string:

```
var currencies = new Dictionary<string, string>
{
    ["USA"] = "USD",
    ["Great Britain"] = "GBP",
    ["Poland"] = "PLN"
};
var currencyInGreatBritain = currencies["Great Britain"];
```

We can define our own indexers in the types we created. Let's define a simple class that works as a wrapper for an array:

```
2 references
class MyList<T>
{
    private T[] _numbers;

    1 reference
    public MyList(T[] numbers)
    {
        _numbers = numbers;
    }
}
```

For now, when trying to use an indexer on an object of this class, we will get a compilation error, because this class does not support it:

```
var myList = new MyList<int>(numbers);
var secondValue = myList[1];
```

To make it work, we must define an indexer accepting an int in the MyList class:

```
public T this[int index]
{
    get => _numbers[index];
    set => _numbers[index] = value;
}
```

On **get**, this indexer simply retrieves the value from the array, and on **set** it overwrites it with the provided value.

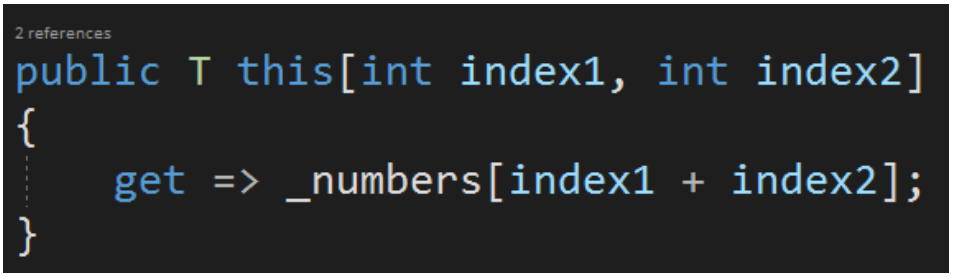
We can define as many indexers as we want if they only differ by types or count of parameters. For example, I can add an indexer accepting a string to this class:

```
public T this[string textIndex]
{
    get => _numbers[int.Parse(textIndex)];
    set => _numbers[int.Parse(textIndex)] = value;
}
```

Indexers with multiple parameters are also allowed:

```
1 reference
public T this[int index1, int index2]
{
    get => _numbers[index1 + index2];
    set => _numbers[index1 + index2] = value;
}
```

Please note that it is possible to have an indexer with getter only (or with setter only, but this is more unusual).



In this case, we will be able to access an element at a given index, but we won't be able to overwrite it.

Let's sum up. Indexers allow instances of a type to be indexed just like arrays. In this way, they resemble properties except that they take parameters. For example, a Dictionary<string, int> has an indexer that allows calling "dictionaryVariable["some key"]" to access the value under some key.

Indexers are most often used with types representing collections, but we can add them to any type. Indexers are simple and natural to use for developers, and we should consider adding them to our types, especially if we already have some methods like "GetValueAtIndex". In this case, we should definitely consider refactoring and introducing an indexer.

Bonus questions:

- "**Is it possible to have a class with an indexer accepting a string as a parameter?**"
  Yes. We can define indexers with any parameters. An example of such a class can be a Dictionary<string, int> as we access its elements like "dict["abc"]".

- "**Can we have more than one indexer defined in a class?**"
  Yes. Just like with method overloading, we can have as many indexers as we want, as long as they differ by the type, count, or order of parameters.