25. What is an ArrayList?

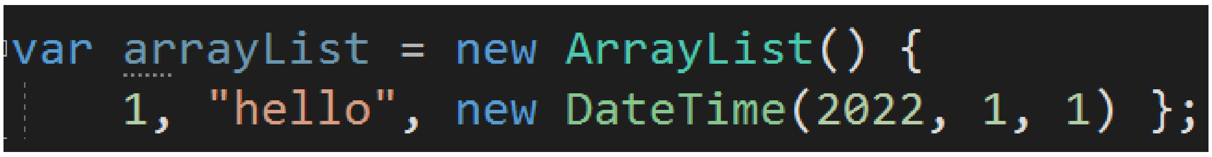
Brief summary: An ArrayList is a collection that can store elements of any type, as it considers them all instances of the System.Object. ArrayLists were widely used in older versions of C#, where the generics were not yet available. Nowadays they should not be used, as their performance is impacted by the fact that they need to box value types.

Let me take you on a journey back in time. A long, long time ago (before 2006) .NET was still at version 1. It was still a pretty new framework (its initial release was in 2002). The C# language itself did not look much as it does now.

At this version of .NET, there was no such thing as **generics**. If you wanted to have a collection of numbers and a collection of strings, arrays were your best choice. You couldn't count on things like the generic List<T> that can hold any type of items.

But as we learned in the lecture "What is an Array?", arrays can be pretty awkward. They have fixed sizes, they also don't provide any convenient methods like Add or Remove. In other words, with arrays only, creating a complete, efficient application that met some real business needs could have been a pain in the neck.

Luckily, there was another way than using plain arrays. The **ArrayList** type. An ArrayList is dynamic a collection, so a collection we can resize, that can hold any type of items. And just to be clear - at the same time. Single ArrayList can hold ints, strings, objects, DateTimes, and anything we want.



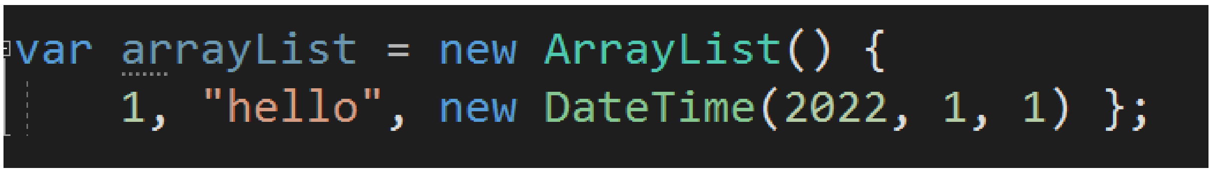
```
var arrayList = new ArrayList() {
    1, "hello", new DateTime(2022, 1, 1) };
```

In statically typed languages like C# this is at least weird. But how does it work? Well, the ArrayList simply treats everything it holds as instances of **System.Object** type. After all, everything in C# can be considered an Object, because every type is derived from the Object class. But there is a problem: if the item we want to store in ArrayList is of a **value type**, it must be **boxed** to be treated as Object, which is a reference type. Boxing is not a cheap operation - it requires moving the value from the stack to the heap and creating a reference for it. Also, at some point, we will need to unbox this item to access the underlying value.

We will talk more about the performance of the Array list later in this lecture.

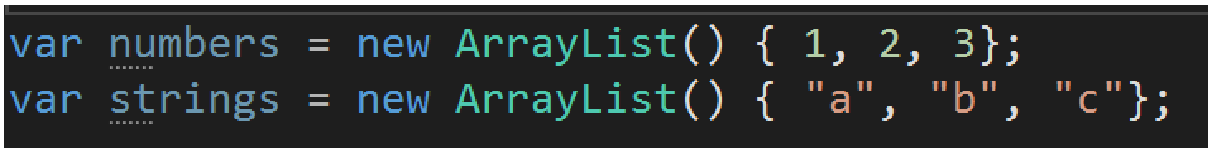
Since ArrayList can hold any type of elements, we don't really know what they are and how can we use them. If I have a List<int>, I know I can, for example, calculate the sum of them. If I have a List<string> I know I can concatenate them. But what can I do with elements of an ArrayList?

The truth is, ArrayList was almost never used like this:



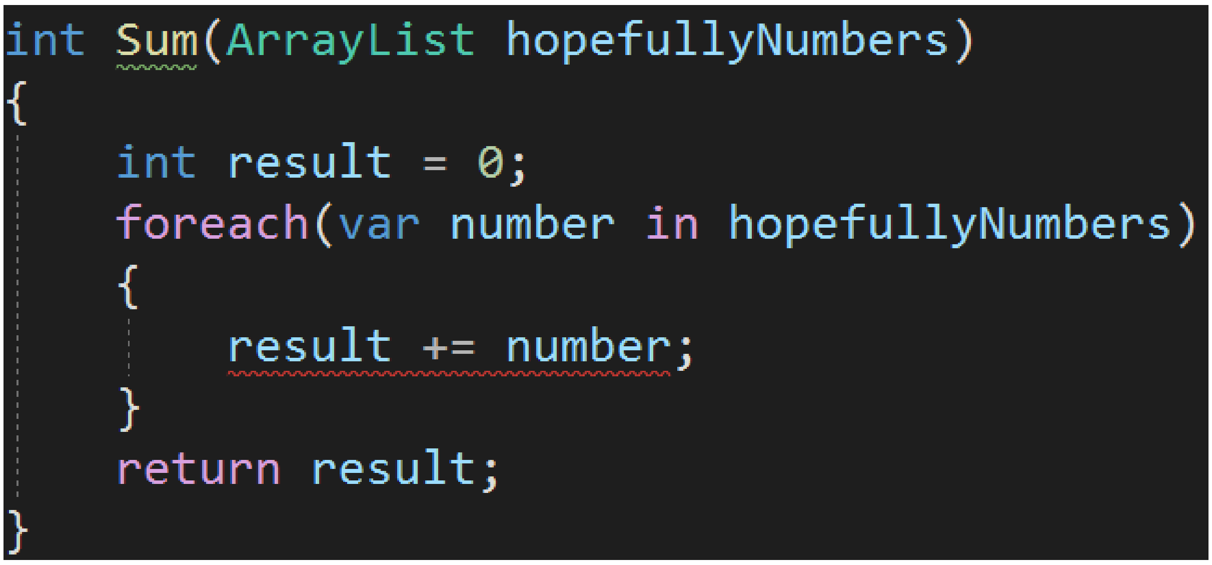
```
var arrayList = new ArrayList() {
    1, "hello", new DateTime(2022, 1, 1) };
```

In most practical cases, it was holding elements of the same type.



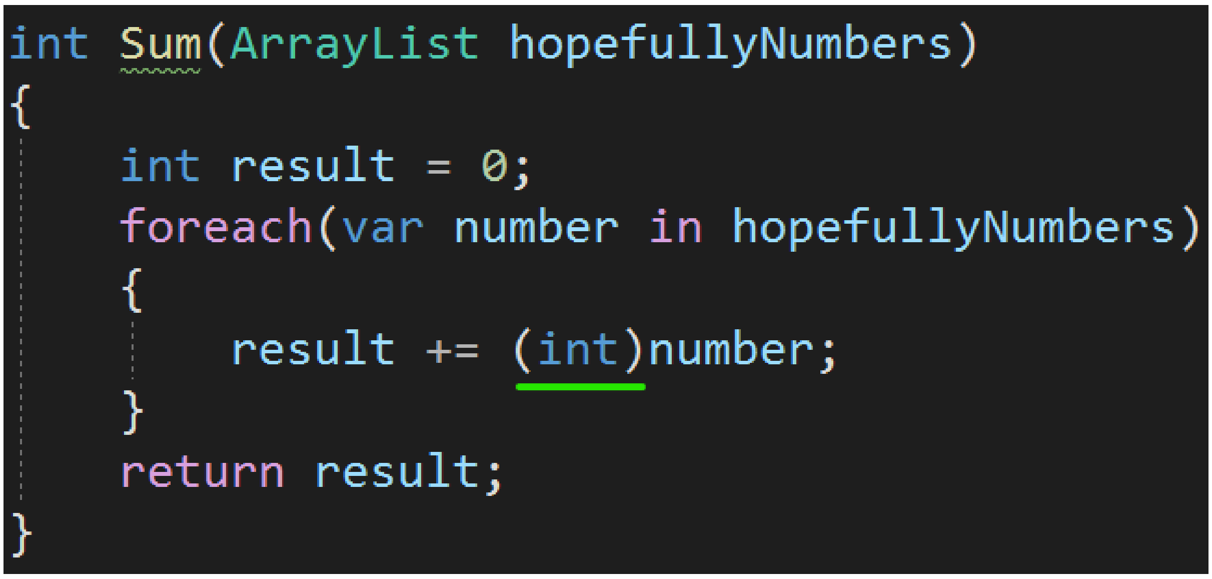
```
var numbers = new ArrayList() { 1, 2, 3};
var strings = new ArrayList() { "a", "b", "c"};
```

That looks "almost" like generic Lists, which again, were not present in .NET before version 2. But those variables are very problematic. Let's say I want to create a method that calculates the sum of elements in a collection of numbers:



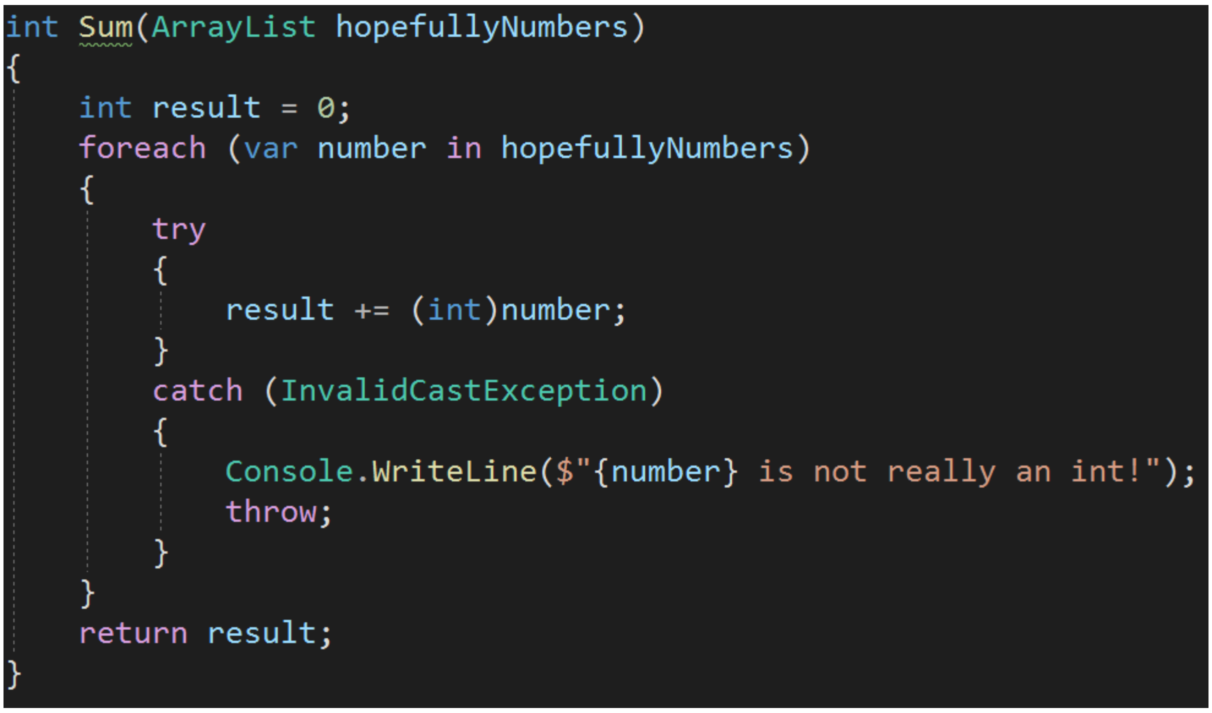
```
int Sum(ArrayList hopefullyNumbers)
{
    int result = 0;
    foreach(var number in hopefullyNumbers)
    {
        result += number;
    }
    return result;
}
```

This doesn't compile. Each element of the ArrayList is an Object, so I can't simply add it to the result. I must first cast it and hope that it will succeed:



```
int Sum(ArrayList hopefullyNumbers)
{
    int result = 0;
    foreach(var number in hopefullyNumbers)
    {
        result += (int)number;
    }
    return result;
}
```

Just to be sure, let's handle the InvalidCastException in this method:



```
int Sum(ArrayList hopefullyNumbers)
{
    int result = 0;
    foreach (var number in hopefullyNumbers)
    {
        try
        {
            result += (int)number;
        }
        catch (InvalidCastException)
        {
            Console.WriteLine($"{number} is not really an int!");
            throw;
        }
    }
    return result;
}
```

As you can see, we were forced to create a lot of code that would not be needed if we knew what types exactly do we deal with. In other words, if we were given a List<int> instead of an ArrayList.

So we know the first big disadvantage of ArrayList - we don't know what is stored inside, so we must be ready for a lot of casting and error handling. The other disadvantage is the performance that I mentioned before - when storing value types in ArrayList, they must all be boxed which can impact the performance very much.

So in this case, the natural question is “When to use ArrayLists over Lists?”. Well, the answer is - never. Unless for some reason you must work in applications written in .NET 1, but I honestly hope that you don’t. Even if you do, consider upgrading the version of .NET rather than working in this ancient technology.

You may then ask, why do we learn about this, if it’s not a big deal since 2006. First of all, the questions about ArrayLists are quite liked by interviewers, as they can be a prelude to a discussion about static and dynamic typing, which we learned more about in the lecture about the “dynamic” keyword.

Secondly, as much as I hope you don’t need to work with ArrayLists, it may turn out that you’ll have to work with some legacy code that still uses them, and then it’s important that you know what you are dealing with. Also, only recently (in 2022) I’ve been working on a brand-new application where someone was using ArrayLists for reasons they couldn’t explain, and as it turned out, it was mostly storing value types. Changing them to Lists not only made the development process much easier, as the neverending casts and error handling could be omitted, but it also made the application over 25% faster.

There is one more case when using ArrayLists may seem tempting - when you *actually* need to store elements of different types in a single collection. But even then, it’s better to use a List<object> as it provides more functionality than ArrayList and will most likely be more consistent with the rest of the application.

Bonus questions:

- "**What is the difference between an array, a List, and an ArrayList?**"
  An array is a basic collection of fixed size that can store any declared type of elements. The List is a dynamic collection (it means, its size can change over time) that is generic, so it can also store any declared type of elements. An ArrayList is a dynamic collection that can store various types of elements at the same time, as it treats everything it stores as instances of the System.Object type.

- "**When to use ArrayList over a generic List<T>?**"
  Never, unless you work with a very old version of C#, which did not support generics. Even if you do, you should rather upgrade .NET to a higher version than work with ArrayLists.