32. Why does string behave like a value type even though it is a reference type?

Brief summary: String is a reference type with the value type semantics. All strings are immutable, which means when they seem to be modified, actually, a new, altered string is created. String has value-type semantics as this is more convenient for developers, but it can't be a value type because string objects can be large, and value types are stored on the stack which has a limited size.

In C#, there is a big difference in how value types and reference types behave. Let's see this difference in code:

```
var valueType = 5;
Console.WriteLine($"Value type is {valueType}");
Console.WriteLine("Executing AddOneToValueType method");
AddOneToValueType(valueType);
Console.WriteLine($"Now value type is {valueType}");
```

```
void AddOneToValueType(int valueType)
{
    ++valueType;
}
```

In this example, we have a variable of type int, which is a **value type**. It is passed to a method that increments it. Because integers are value types, when they are passed as parameters to methods, a copy of the value is created. That's why after this method is executed, the value of the original variable will not be changed.

```
Value type is 5
Executing AddOneToValueType method
Now value type is 5
```

Let's consider a similar example now, but with **reference types**:

```
var referenceType = new List<int> { 5 };
Console.WriteLine($"Reference type has {referenceType.Count} elements");
Console.WriteLine("Executing AddOneToReferenceType method");
AddOneToReferenceType(referenceType);
Console.WriteLine($"Now reference type has {referenceType.Count} elements");
```

```
void AddOneToReferenceType(List<int> referenceType)
{
    referenceType.Add(6);
}
```

A List<int> is a reference type. It is passed to a method by reference, which means inside the method we add an element to the original List<int> object. That's why after the method is executed, the count of elements in the list is 2:

```
Reference type has 1 elements
Executing AddOneToReferenceType method
Now reference type has 2 elements
```

Now, let's see the last example. This time we will deal with **strings**:

```
var text = "Hello!";
Console.WriteLine($"text is {text}");
Console.WriteLine("Executing AddOneToString method");
AddOneToString(text);
Console.WriteLine($"Now text is {text}");
```

```
void AddOneToString(string text)
{
    text += "1";
}
```

And now, let's see the result:

```
text is Hello!
Executing AddOneToString method
Now text is Hello!
```

The string has **not** been modified. It's the same behavior we've seen for value types. But here is the plot twist: **string is a reference type** in C#! So what is going on?

To understand it, we must first realize, that under the hood **string is an array of chars**. As we learned in the "What is an Array?" lecture, arrays are collections of fixed size. Once an array is created, its size never changes. If we want to add an element to an array, we must declare a new, bigger array, copy the old array to it, and set the value under the last index to the new element. And that's exactly what happens when we modify a string. It's not really changing the original string. It is creating a new string which then gets assigned to the variable:

```
var text = "abc";
text += "1";
```

In this case, a new string gets created, containing the original string with "1" added to the end.

This means strings in C# are **immutable**. A string object is never modified. Even if it seems like it, a new object is actually created under the hood.

As we already know, when passed as a parameter to a method, the string behaves like a value type. But this is not the end of similarities of string to value types. Also, its == operator is overloaded, so it compares strings by value, not by reference. For reference types, equality is compared by reference, so those two Lists will not be equal, because they are two different objects pointed to by two different references:

```
List<int> list1 = new List<int> { 1, 2, 3 };
List<int> list2 = new List<int> { 1, 2, 3 };
```

On the other hand, this equality comparison will return true, because strings are compared by value, even though they are reference types:

```
var string1 = "abc";
var string2 = "abc";
```

Technical reasons aside, it is actually desired for strings to behave more like value types. I think most programmers would be surprised if for the above strings the equality check with the == operator would return false as it should for regular reference types. Also, it would be pretty challenging if the modification of a string in a method to which it was passed as a parameter would affect the original string object. In general, people tend to think of strings in a similar way as they think of value types, and learning to use them as other reference types would most likely make C# quite disliked in the programming community.

You may wonder: since string behaves like a value type, why isn't it one? Maybe another design than using the array of characters would be possible, and string could be a value type like numbers or DateTime?

Well, the answer is (as so often) related to performance. Value types are stored on the stack, which has a limited size (1 MB for 32-bit processes and 4MB for 64-bit processes). Strings can be quite huge, and they could simply not fit on the stack. They are stored on the heap instead, along with other reference types.

One more thing before we wrap up. Because strings are immutable, if we have multiple strings of the same value, we can use the optimization called **Interning**. **Interning** means that if multiple string variables hold strings that are known to be equal, the runtime actually points their references to a single string object, thereby saving memory. This optimization wouldn't work if strings were mutable, because then changing one string variable would have unpredictable results on other string variables.

Let's summarize. String is a reference type with the value type semantics. All strings are immutable, which means when they seem to be modified, actually, a new, altered string is created. String has value-type semantics as this is more convenient for developers, but it can't be a value type because string objects can be large, and value types are stored on the stack which has a limited size.

Because this copy-and-alter way of modifying strings can be performance-costly, it is recommended to use the StringBuilder class when building strings incrementally. We will learn more about it in the next lecture.

Bonus questions:

- "**What is interning of strings?**"
  Interning means that if multiple strings are known to be equal, the runtime can just use a single string, thereby saving memory. This optimization wouldn't work if strings were mutable, because then changing one string would have unpredictable results on other strings.

- "**What is the size of the stack in megabytes?**"
  It's 1 MB for 32-bit processes and 4 MB for 64-bit processes.

- "**What is the underlying data structure for strings?**"
  It's an array of chars. Arrays by definition have fixed size, which is a reason why strings are immutable - we couldn't modify a string by adding new characters to it, because they wouldn't fit in the underlying array.