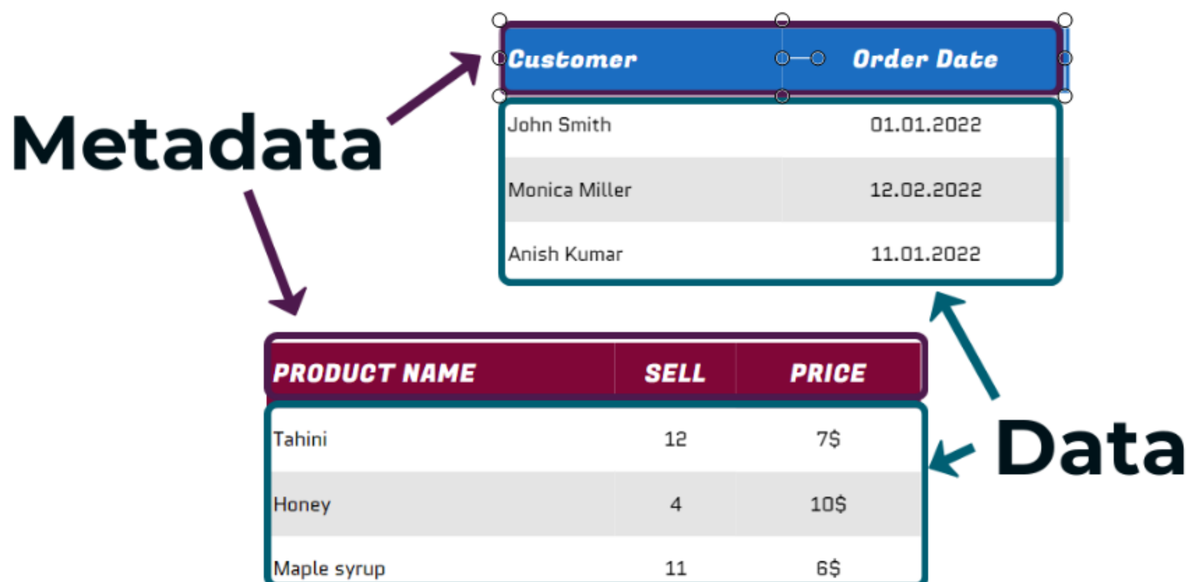
17. What are attributes?

Brief summary: Attributes add metadata to a type. In other words, they are a way to add information about a type or method to the metadata which describes that type or method.

Attributes add metadata to a type. In other words, they are a way to add information about a type or method to the existing metadata which describes that type or method, which we can read from the Type object.

First, let's understand what **metadata** is. Generally speaking, metadata is data providing information about other data. For example, when working with databases, the data stored inside the database is the actual data, while the structure of tables and relations between them is metadata.



In programming, metadata describes types used in an application.

First, let's consider this simple class:

```
public class Person
{
    public string Name { get; }
    public int YearOfBirth { get; }

    public Person(string name, int yearOfBirth)
    {
        Name = name;
        YearOfBirth = yearOfBirth;
    }

    public Person(string name) => Name = name;
}
```

There is a lot of metadata describing this class. For example, the metadata contains the information that this class is named “Person”, it is public, non-static, non-sealed, etc. It contains two get-only public properties called Name and YearOfBirth. It has one public constructor taking two string parameters, and one taking one parameter. The **actual data** stored in an instance of this class would be the string representing the name, and int representing the year of birth.

We can access all the class’s metadata at runtime using **reflection**, which we learned about in the previous lecture.

```
var type = typeof(Person);
```

Sometimes we want to add extra metadata to a type or member, and this is what **attributes** are for.

Let’s consider the following example. We want to have a common way of validating some data in the application. No matter the type, we want to be able to specify that its members of the string type must have a certain length. This is how it would look like:

```
1 reference
public class Dog
{
    1 reference
    public string Name { get; } //length must be between 2 and 10
    0 references
    public Dog(string name) => Name = name;
}
5 references
public class Person
{
    2 references
    public string Name { get; } //length must be between 2 and 25
    1 reference
    public int YearOfBirth { get; }
```

I want the Validator class to be able to take objects of **any class** and check if for any of their properties this validation is required. If so, it should check if the values of those properties are valid.

```
var validPerson = new Person("John", 1982);
var invalidDog = new Dog("R");
var validator = new Validator();

Console.WriteLine(validator.Validate(validPerson) ?
    "Person is valid" :
    "Person is not valid");
Console.WriteLine(validator.Validate(invalidDog) ?
    "Dog is valid" :
    "Dog is not valid");
```

All right. So what we want to do is to add some metadata to the Name properties in both Person and Dog types defining their minimal and maximal lengths. This is some “extra” metadata and to define it we must use a custom attribute. This is how it should look like:

```
2 references
public class Dog
{
    [StringLengthValidate(2, 10)]
    1 reference
    public string Name { get; }
    1 reference
    public Dog(string name) => Name = name;
}
4 references
public class Person
{
    [StringLengthValidate(2, 25)]
    2 references
    public string Name { get; }
```

As you can see, to add an attribute to a member or type, we simply must write its name in the brackets above the type or member we want to add it to. As you can see, the attribute we have requires two parameters - minimal and maximal length. Now, let's define the StringLengthValidateAttribute class.

```
class StringLengthValidateAttribute : Attribute
{
    3 references
    public int Min { get; }
    3 references
    public int Max { get; }

    2 references
    public StringLengthValidateAttribute(int min, int max)
    {
        Min = min;
        Max = max;
    }
}
```

All attributes must derive from the **Attribute** base class. Also, typically their names end with "Attribute". As you saw before, this postfix is omitted when we actually use the attribute:

```
[StringLengthValidate(2, 25)]
2 references
public string Name { get; }
```

One more thing. We can also define what the attribute can be applied to. In our case we want it to be applied to properties. To enforce that, we must actually use a built-in attribute called AttributeUsage:

```
[AttributeUsage(AttributeTargets.Property)]
7 references
class StringLengthValidateAttribute : Attribute
```

Great. Now all left to do is to define the Validator class.

```
class Validator
{
        2 references
        bool Validate(object obj)
```

This class will simply contain Validate method which can take any object. For this object, we will look for its properties with the StringLengthValidateAttribute defined.

```
bool Validate(object obj)
{
    var type = obj.GetType();
    var propertiesToValidate = type
        .GetProperties()
        .Where(property =>
            Attribute.IsDefined(
                property, typeof(StringLengthValidateAttribute)));
```

As you can see we selected the properties for which this attribute is defined using the LINQ's Where method along with Attribute.IsDefined method. Now, we can iterate those properties and check if their lengths are correct. But first, we must make sure that the property is a string. If not, we want to throw an exception, because it means that a developer added this attribute to a different type by mistake:

```
foreach (var property in propertiesToValidate)
{
    var attribute = (StringLengthValidateAttribute)
        property.GetCustomAttributes(
            typeof(StringLengthValidateAttribute), true).First();
    object? propertyValue = property.GetValue(obj);
    if(propertyValue is not string)
    {
        throw new InvalidOperationException(
            $"Attribute {nameof(StringLengthValidateAttribute)} " +
            $"can only be applied to strings.");
    }
```

Otherwise, we can validate the value:

```
if(propertyValue is not string)
{
    throw new InvalidOperationException(
        $"Attribute {nameof(StringLengthValidateAttribute)} " +
        $"can only be applied to strings.");
}
var value = (string)propertyValue;
if (value.Length < attribute.Min || value.Length > attribute.Max)
{
    Console.WriteLine($"Property {property.Name} is invalid. " +
        $"Value: {value}. The length must be between " +
        $"{attribute.Min} and {attribute.Max}");
    return false;
}
```

And this is the whole method:

```
bool Validate(object obj)
{
    var type = obj.GetType();
    var propertiesToValidate = type
        .GetProperties()
        .Where(property =>
            Attribute.IsDefined(
                property, typeof(StringLengthValidateAttribute)));

    foreach (var property in propertiesToValidate)
    {
        var attribute = (StringLengthValidateAttribute)
            property.GetCustomAttributes(
                typeof(StringLengthValidateAttribute), true).First();
        object? propertyValue = property.GetValue(obj);
        if(propertyValue is not string)
        {
            throw new InvalidOperationException(
                $"Attribute {nameof(StringLengthValidateAttribute)} " +
                $"can only be applied to strings.");
        }
        var value = (string)propertyValue;
        if (value.Length < attribute.Min || value.Length > attribute.Max)
        {
            Console.WriteLine($"Property {property.Name} is invalid. " +
                $"Value: {value}. The length must be between " +
                $"{attribute.Min} and {attribute.Max}");
            return false;
        }
    }
     return true;
}
```

All right. Let's make sure it works:

```
Person is valid

Property Name is invalid. Value: R. The length must be
 between 2 and 10
Dog is not valid
```

Great! That's what we wanted.

As you can see attributes can be quite powerful. They are widely used in native .NET classes, as well as external libraries. For example, if you ever used NUnit, you must have used some of its attributes:

```
[TestFixture]
public class CalculatorTests
{
    private Calculator? _cut;

    [SetUp]
    public void SetUp()
    {
        _cut = new Calculator();
    }

    [Test]
    public void Add5And10_ShallGive15()
    {
        Assert.AreEqual(15, _cut!.Add(10, 5));
    }
```

Let's summarize. Attributes add metadata to a type. In other words, are a way to add information about a type or method to the metadata which describes that type or method. To add an attribute to a member or type, we simply must write its name in the brackets above the type or member we want to add it to. There are plenty of built-in Attributes in C# standard library, but we can also create attributes of our own, simply by creating classes derived from the Attribute base class.

Bonus questions:

- "**What is metadata?**"
  Generally speaking, metadata is data providing information about other data. For example, when working with databases, the data stored inside the database is the actual data, while the structure of tables and relations between them is metadata. In programming, metadata describes types used in an application. We can access it in the runtime using reflection, to get the information about some type, for example, what methods or what constructors it contains.

- "**How to define a custom attribute?**"
  To define a custom attribute we must define a class that is derived from the Attribute base class.