31. What are records and record structs?

Brief summary: Records and record structs are new types introduced in C# 9 and 10. They are mostly used to define simple types representing data. They support value-based equality. They make it easy to create immutable types.

Important: records are available since C# 9. Record structs are available since c# 10.

When programming, we often need to define simple data structures, that don't really hold any business logic - they simply store data. Let's define a Point class.

```
public class PointClass
{
    1 reference
    public int X { get; }
    1 reference
    public int Y { get; }

    0 references
    public PointClass(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

This class only holds two integers X and Y. I want objects of this class to be **immutable**, so once set, they will not be updated. That's why I only added getters to the X and Y properties. Setters are not available.

I would like the objects of this class to be **nicely printed.** Now, if I call Console.WriteLine(somePoint) I will get the full type name printed to the console, so "Namespace.PointClass". This is not very convenient. Let's override the ToString method:

```
public override string ToString()
{
    return $"X:{X}, Y:{Y}";
}
```

Now, something like “X:10, Y:5” will be printed.

Next, I would like to use objects of the Point class to as **keys in the Dictionary**. Currently, since Point is a class, its objects are compared by reference. That means, even if I have two points equal by value, they will be considered two different keys in a Dictionary:

```
var somePoint = new PointClass(10, 5);
var otherPointEqualByValue = new PointClass(10, 5);
var dict = new Dictionary<PointClass, string>();
dict[somePoint] = "aaa";
dict[otherPointEqualByValue] = "bbb";
```

After this code is executed, the Dictionary will have a size of **two**, because each point is considered a different key, as they differ by reference. I would like other behavior - if two points are equal by value, they are considered the same key by the Dictionary. To achieve this, I must overwrite the GetHashCode and Equals methods:

```
0 references
public override int GetHashCode()
{
    return HashCode.Combine(X, Y);
}

2 references
public override bool Equals(object? obj)
{
    return obj is PointClass && Equals((PointClass)obj);
}
```

Now I also must provide the Equals method that accepts a **PointClass** object, not an **object**. That means, my PointClass shall implement the IEquatable<PointClass> interface:

```
public bool Equals(PointClass? other)
{
    return other is not null && other.X == X && other.Y == Y;
}
```

Let's see the whole class:

```
public class PointClass : IEquatable<PointClass>
{
    3 references
    public int X { get; }
    3 references
    public int Y { get; }

    2 references
    public PointClass(int x, int y)
    {
        X = x;
        Y = y;
    }

    0 references
    public override string ToString()
    {
        return $"X:{X}, Y:{Y}";
    }

    0 references
    public override int GetHashCode()
    {
        return HashCode.Combine(X, Y);
    }

    0 references
    public override bool Equals(object? obj)
    {
        return obj is PointClass && Equals((PointClass)obj);
    }

    2 references
    public bool Equals(PointClass? other)
    {
        return other is not null && other.X == X && other.Y == Y;
    }
}
```

Well… it works, but it's a lot of code, and all of it only to implement a simple data structure that:

- prints itself nicely

- is compared by value
- provides custom GetHashCode and Equals methods implementations so it can safely be used in hashed collections

At some point, the creators of C# realized that this is a common issue. As a solution, they introduced **records**. Before I explain exactly what records are, let me show you how exactly the same behavior as we defined in the type above can be achieved with records:

```
public record PointRecord(int X, int Y);
```

That's it. Only one line of code, and it does the same thing as 31 lines of code we needed to define the PointClass.

Records are new types, joining classes and structs. They are available starting with C# 9. Let's list the most important information about records:

- records are reference types
- …but they base on value-type equality, which means, two records with identical values of properties will be considered equal even if they differ by reference
- like classes, they support inheritance
- the compiler generates the following methods for records:
    - an override of Equals(object?) method
    - a virtual Equals(ThisRecord?) method (this method comes from the IEquatable<ThisRecord> interface which records implement)
    - and override for the GetHashCode method
    - overloads of == and != operators
    - an override of the ToString method, which prints the names of the properties with their values

The record we defined above is even more special: it's a so-called **positional record**, so a record that doesn't even have a body. Later in the article, we will learn how to define non-positional records. For now let's just note that for positional records, the compiler also generates:

- a primary constructor whose parameters match the positional parameters on the record declaration
- public properties for each parameter of a primary constructor. Those properties are read-only (but they are **not** for **record structs**, which we will learn about a bit later)
- a Deconstruct method to extract properties from the record

All right. Let's see a regular, non-positional record now:

```
public record PointNonPositionalRecord
{
    2 references
    public int X { get; set; } //non-positional records can be read-write
    3 references
    public int Y { get; set; }

    1 reference
    public PointNonPositionalRecord(int x, int y)
    {
        X = x;
        Y = y;
    }

    0 references
    public int Sum() //we can add methods to records
    {
        return X + Y;
    }
}
```

As you can see we need to write a little more code (like explicitly defining the properties and the constructor) but as a reward, we can add methods or make the properties writable. Remember that methods like GetHashCode, Equals, or ToString are still generated by the compiler and we don't need to worry about them.

As we learned in the last lecture, the **immutability** of types is a desired trait. Records are perfect for representing immutable types. To make things even easier, they provide non-destructive mutation implemented with the **with** keyword. This may sound cryptic, so let's see an example. Let's say I have some point, and I want to update its Y property:

```
var somePointRecord = new PointRecord(2, 3);
var somePointBythNewY = somePointRecord with { Y = 6 };
```

With the **with** keyword, I created a **new** Point equal to the old one, but with Y set to the new value of 6. The old point is immutable, so it cannot be changed. I can change as many properties as I want with the "with" keyword.

Starting with C# 10, **record structs** were introduced. They are similar to records, with some differences:

- they are value types
- positional record structs are read-write by default, which means their properties are mutable
- record structs can be declared as readonly, making them immutable

```
public readonly record struct PointReadonlyRecordStruct(int X, int Y);
```

- For record structs, the compiler also generates a parameterless constructor which sets all its properties to the default values

When deciding whether to use records or record structs, you should take the same things into consideration as when deciding whether to use classes or structs. In general - if the type is simple and you want value-type behavior like passing parameters by value, you should go for the record structs.

Also, records and record structs support deconstruction. We learned more about it in the "What is deconstruction?" lecture.

```
var somePointRecord = new PointRecord(2, 3);
var somePointBythNewY = somePointRecord with { Y = 6 };
var (localX, localY) = somePointBythNewY;
```

Let's summarize. Records and record structs are new types introduced in C# 9 and 10. They are mostly used to define simple types representing data. They support value-based equality. They make it easy to create immutable types.

Bonus questions:

- "**What is the purpose of the "with" keyword?**"
  The "with" keyword is used to create a copy of a record object with some properties set to new values. In other words, it's used to perform a non-destructive mutation of records.

- "**What are positional records?**"
  Positional records are records with no bodies. The compiler generates properties, constructor, and the Deconstruct method for them. They are a shorter way of defining records, but we can't add custom methods or writable properties to a positional record.