Records

Syntax & usage

Record

A **Record** is a new reference type introduced in C# 9.

- provides built-in functionality for encapsulating data
- saves a developer from writing boilerplate code
- can be mutable, but primarily intended for supporting immutable data models
- on the IL level, it's compiled to a class

Record

The **`record`** type offers the following features:

- A short and concise syntax for creating type with immutable properties
- Built-in behavior for a data-driven reference type:
  - Value equality
  - Syntax for nondestructive mutation
  - Built-in formatting for display
- Support for inheritance hierarchies

Positional records

In the previous module, we finished creating the **SpeakerDTO** by introducing the **init** only setters.

```
public class SpeakerDTO
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

We can convert this type into the **record type** by using the following code:

```
public record SpeakerDTO(string FirstName, string LastName);
```

To create and use an instance of the record, we can use an automatically generated constructor and deconstruct method:

```
var speaker = new SpeakerDTO("Konrad", "Kokosa"); // positional construction
var (firstName, lastName) = speaker;              // positional deconstruction
```

This "inline" style of declaring a record is called a **positional**, so our record can be called **positional record** or a record with **positional parameters**.

Positional records

When you decide to define properties using the positional syntax, the compiler generates:

- **`public constructor`** with parameters matching the record declaration
- **`public init-only`** property for each positional parameters
- A **`Deconstruct`** method with an out parameter for each positional parameter provided in the record declaration. This method is generated only if there is more than one positional parameter.
- and more features, that will be described in the next slides

Positional records

If the generated property definition doesn't meet your requirements, you can define your own property of the same name. When you do it, the generated methods will use your property definition for the operations. For instance, the following piece of code makes the **`FirstName`** property internal instead of public.

```
public record SpeakerDTO(string FirstName, string LastName)
{
    internal string FirstName { get; init; } = FirstName;
}
```

Non-positional records

A record can also have properties declared with standard property syntax. Sometimes these kinds of records are called **non-positional**.

```
public record SpeakerDTO
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

Combined syntax

You can combine the approach and use **positional parameters** and **default property syntax** together, for example:

```
public record SpeakerDTO(string FirstName, string LastName)
{
    public bool IsMicrosoftMVP { get; init; }
}

public static void Main()
{
    var speaker = new SpeakerDTO("Konrad", "Kokosa")
    {
        IsMicrosoftMVP = true
    };

    Console.WriteLine(speaker.FirstName); // Konrad
}
```

Combined syntax

The **record** type is designed to work well with combined syntax. Most of the features will work no matter where the property is declared, except for two small edge cases.

When a property is declared with default syntax, it has no support for construction and deconstruction. It means that it **doesn't exist in the generated constructor and generated deconstruct method**, but we can implement it manually

```
public record SpeakerDTO(string FirstName, string LastName)
{
    public bool IsMicrosoftMVP { get; init; }

    public SpeakerDTO(string firstName, string lastName, bool isMicrosoftMvp) : this(firstName, lastName)
    {
        IsMicrosoftMVP = isMicrosoftMvp;
    }

    public void Deconstruct(out string firstName, out string lastName, out bool isMicrosoftMvp)
    {
        firstName = FirstName;
        lastName = LastName;
        isMicrosoftMvp = IsMicrosoftMVP;
    }
}
```

Empty records

Similar to the class without properties, a record can also be declared as an empty type without any field and properties.

```
public record SpeakerDTO();

// or

public record SpeakerDTO
{
}
```