Records

immutability and equality

Immutability

The **`Record`** type offers immutability by default for properties declared with positional parameters and for properties created manually with default property syntax that has **`init only`** setter.

However, a record is not necessarily immutable. We can declare properties with **`set`** accessors together with fields that are not readonly.

Immutability

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

    public bool RequiresOnlineSession { get; set; }
}

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

    speaker.RequiresOnlineSession = CheckTravelAvailability();
}
```

Value equality

Value equality means that two variables of the `record` type are equal if the types match and all property and field values match

For other reference types, like classes, **equality** means **identity**, for example, two variables of a reference type are equal if they point to the same object. Because of this difference, records cannot be applied in some scenarios.

For example, the *Entity Framework* depends on the reference equality to manage entities. Because of the value equality that record provides, records are not appropriate for entities tracked by Entity Framework.

Value equality

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

public static void Main()
{
  var speaker1 = new SpeakerDTO("Konrad", "Kokosa", true);
  var speaker2 = new SpeakerDTO("Konrad", "Kokosa", true);
  Console.WriteLine(speaker1 == speaker2); // true;
  Console.WriteLine(speaker1.Equals(speaker2)); // true;
  Console.WriteLine(ReferenceEquals(speaker1, speaker2)); // false;

  var speaker3 = new SpeakerDTO("Szymon", "Kulec", true);
  Console.WriteLine(speaker1 == speaker3); // false;
}
```

Shallow equality

Please keep in mind that a **record** type can also contain other reference types, for example arrays or custom objects.

In thecode below we have a record with an array of integers. In such case, the equality comparer relies on the underlying reference to the array. If both record instances have properties pointing to different objects, they are not equal.

```
public record SpeakerDTO(string FirstName, string LastName, int[] Scores);

public static void Main()
{
  var a = new SpeakerDTO("Szymon", "Kulec", new[] { 5, 4, 5 });
  var b = new SpeakerDTO("Szymon", "Kulec", new[] { 5, 4, 5 });

  Console.WriteLine(a == b); // false

  int[] scores = a.Scores;
  var c = new SpeakerDTO("Szymon", "Kulec", scores);
  var d = new SpeakerDTO("Szymon", "Kulec", scores);

  Console.WriteLine(c == d); // true
}
```

Value equality

To provide such functionality, C# compiler generates following code:

- an override of `Equals(object? obj)`
- A `virtual Equals(SpeakerDTO? other)` which method implements `IEquatable<T>`
- an override of `GetHashCode()`
- overrides of operators `==` and `!=`

Value equality - generated code

For the simplification, let's declare DTO with only one property:

```
public record SpeakerDTO(string FirstName);
// compiles into the same output IL as the following
public class SpeakerDTO : IEquatable<SpeakerDTO>
{
      [CompilerGenerated]
      [DebuggerBrowsable(DebuggerBrowsableState.Never)]
      private readonly string <FirstName>k__BackingField;

      public string FirstName
      {
        [CompilerGenerated]
        get
        {
          return <FirstName>k__BackingField;
        }
        [CompilerGenerated]
        init
        {
          <FirstName>k__BackingField = value;
        }
      }
      [...]
}
```

Value equality - generated code

```
public class SpeakerDTO : IEquatable<SpeakerDTO>
{
    protected virtual Type EqualityContract
    {
      [CompilerGenerated]
      get => typeof(SpeakerDTO);
    }

    public override bool Equals(object? obj) => Equals(obj as SpeakerDTO);

    public virtual bool Equals(SpeakerDTO? other)
    {
      return (object)other != null &&
      EqualityContract == other!.EqualityContract &&
      EqualityComparer<string>.Default.Equals(FirstName, other!.FirstName);
    }

    public static bool operator !=(SpeakerDTO? r1, SpeakerDTO? r2) => !(r1 == r2);
    public static bool operator ==(SpeakerDTO? r1, SpeakerDTO? r2) => (object)r1 == r2 || (r1?.Equals(r2) ?? false);

    public override int GetHashCode()
    {
      return EqualityComparer<Type>.Default.GetHashCode(EqualityContract) *
        -1521134295 + EqualityComparer<string>.Default.GetHashCode(FirstName);
    }

    [...]
}
```

Value equality for classes

To achieve the same behavior for the class, we would have to create the same overrides for equality methods and operators, however, this process is often time-consuming and error-prone.

Having this functionality built-in to the compiler prevents errors and bugs that would appear after forgetting to update the custom equality logic code, for example when properties or fields are added or changed.

You can still write your own implementation of any generated method. When a record type has a method that has the same signature as any generated method, the compiler doesn't generate that method.