accu
2022

C# VALUES AND VALUE TYPES

STEVE LOVE

Me

Steve Love

@IAmSteveLove

steve@arventech.com

User defined types

- The family of values
- Why they matter
- Comparing characteristics

The humble struct

```
public struct Colour
{
    public int Red   { get; set; }
    public int Green { get; set; }
    public int Blue  { get; set; }
}
```

```
[Test]
public void Colour_has_value_equality()
{
    var orange = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };
    var text   = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };

    Assert.That(orange, Is.EqualTo(text));
}
```

✓

Reference semantics

```
public class Colour
{
    public int Red   { get; set; }
    public int Green { get; set; }
    public int Blue  { get; set; }
}
```

```
[Test]
public void Colour_has_value_equality()
{
    var orange = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };
    var text   = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };

    Assert.That(orange, Is.EqualTo(text));
}
```

❌

Equals method

```
public class Object
{
    public virtual bool Equals(object? other)
        => other == this;
}
```

```
var orange = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };
var apple = "I am an apple";

Assert.That(orange.Equals(apple), Is.False);
```

✓

Classes as values

```
public class Colour
{
    public int Red   { get; set; }
    public int Green { get; set; }
    public int Blue  { get; set; }

    public override bool Equals(object? obj)
        => obj == this ||
           obj is Colour other &&
           GetType() == other.GetType() &&
           Red == other.Red && Green == other.Green && Blue == other.Blue;

    public override int GetHashCode()
        => HashCode.Combine(Red, Green, Blue);
}
```

```
[Test]
public void Colour_has_value_equality()
{
    var orange = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };
    var text   = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };

    Assert.That(orange, Is.EqualTo(text));
}
```

✓

Values and equality

```
public sealed class Colour : IEquatable<Colour>
{
    public Colour(int r, int g, int b)
        => (Red, Green, Blue) = (r, g, b);

    public int Red   { get; }
    public int Green { get; }
    public int Blue  { get; }

    public bool Equals(Colour? other)
        => (object?)other == this ||
           other is not null &&
           Red == other.Red && Green == other.Green && Blue == other.Blue;

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

    public override int GetHashCode()
        => HashCode.Combine(Red, Green, Blue);
}
```

Convenient equality

```
public static bool operator==(Colour? left, Colour? right)
    => left?.Equals(right) ?? right is null;

public static bool operator!=(Colour? left, Colour? right)
    => !left?.Equals(right) ?? right is not null;
```

```
var orange = new Colour(0xFF, 0xA0, 0);
var text   = new Colour(0xFF, 0xA0, 0);

Assert.That(orange == text, Is.True);
```

✓

Records

```
public sealed record Colour
{
    public Colour(int r, int g, int b)
        => (Red, Green, Blue) = (r, g, b);

    public int Red   { get; }
    public int Green { get; }
    public int Blue  { get; }
}
```

```
var orange = new Colour(0xFF, 0xA0, 0);
var text   = new Colour(0xFF, 0xA0, 0);

Assert.That(orange == text, Is.True);
```

✓

Positional records

```
public sealed record Colour(int Red, int Green, int Blue);
```

Operator ==

```
var orange = new Colour(Red: 0xFF, Green: 0xA0, Blue: 0);
var text   = new Colour(Red: 0xFF, Green: 0xA0, Blue: 0);

Assert.That(orange == text, Is.True);
```

✓

Named properties

```
Assert.That(orange.Red,   Is.EqualTo(text.Red));
Assert.That(orange.Green, Is.EqualTo(text.Green));
Assert.That(orange.Blue,  Is.EqualTo(text.Blue));
```

✓

Non-destructive mutation

```
var green = orange with { Red = 0, Green = 0xFF };

Assert.That(orange, Is.EqualTo(text));
Assert.That(ReferenceEquals(orange, green), Is.False);
```

✓

Value type record structs

```
public readonly record struct Colour(int Red, int Green, int Blue);
```

Object initialization

```
var orange = new Colour(0xFF, 0xA0, 0);
var text   = new Colour { Red = 0xFF, Green = 0xA0, Blue = 0 };
```

The Anemic Domain Model

"The basic symptom of an Anemic Domain Model is that at first blush it looks like the real thing. There are objects, many

named after the nouns in the domain space,

and these objects are connected with the

rich relationships and structure

that true domain models have. The catch comes when you [...] *realize that there is hardly any behavior on these objects, making them*

little more than bags of getters and setters."

Martin Fowler

Values are in the design

Data Transfer Objects are not Domain Objects or Values

...or *vice versa*

Accepting the defaults

```
public readonly record struct Temperature(double Amount);
```

What kind of `Temperature`?

```
var heat = new Temperature(98.6);
```

Defining units

```
public readonly record struct Temperature(double InCelsius)
{
    public double InFahrenheit => InCelsius * 1.8 + 32;

    public static Temperature FromCelsius(double c)
        => new Temperature(c);

    public static Temperature FromFahrenheit(double f)
        => new Temperature((f - 32) / 1.8);
}
```

Explicit units for `Temperature`

```
var heatWave = Temperature.FromFahrenheit(98.6);
var bodyTemp = Temperature.FromCelsius(37);

var hot = bodyTemp.InFahrenheit;
```

Out of range

```
var extraCold = Temperature.FromCelsius(-1000000);
```

Validating values

```
public readonly record struct Temperature
{
    private Temperature(double val)
        => InCelsius = val switch
        {
            < -273.15 => throw new ArgumentOutOfRangeException( /*...*/ ),
            _         => val
        };

    public double InCelsius { get; }

    public static Temperature FromCelsius(double c)
        => new Temperature(c);

    // ...
```

Range validation

```
< -40 or > 500 =>  throw new ArgumentOutOfRangeException( /*...*/ ),
```

Floating point comparisons

```
var heatWave = Temperature.FromFahrenheit(98.6);
var bodyTemp = Temperature.FromCelsius(37);

Assert.That(heatWave, Is.EqualTo(bodyTemp));
```

❌

```
    Expected: 37.0d
      But was:  36.999999999999993d
```

Compiler-generated `Equals`

```
public readonly struct Temperature : IEquatable<Temperature>
{
    // ...

    public bool Equals(Temperature other)
        => EqualityComparer<double>.Default.Equals(InCelsius, other.InCelsius);

    public override int GetHashCode()
        => EqualityComparer<double>.Default.GetHashCode(InCelsius);

    public double InCelsius { get; }

    // ...
}
```

Custom `Equals`

```csharp
public readonly record struct Temperature
{
    // ...

    public bool Equals(Temperature other)
        => Math.Round(Math.Abs(InCelsius - other.InCelsius), 7) == 0;

    public override int GetHashCode()
        => InCelsius.GetHashCode();

    public double InCelsius { get; }

    // ...
}
```

Sorting

```
public readonly record struct Temperature : IComparable<Temperature>
{
    // ...

    public double InCelsius { get; }

    public int CompareTo(Temperature other)
        => InCelsius.CompareTo(other.InCelsius);

    public static bool operator<(Temperature left, Temperature right)
        => left.CompareTo(right) < 0;

    public static bool operator>(Temperature left, Temperature right)
        => left.CompareTo(right) > 0;
}
```

Default initialization

```
var freezing = new Temperature();
```

Perhaps

`sealed record Temperature`

would be better

Defaults work for some types

```
public readonly record struct Colour
    (int Red, int Green, int Blue);

var black = new Colour();
```

Just remember:default-initialized reference properties are `null`

Records are great when...

- there are no floating-point fields
- ...or references which may be `null`
- don’t need to be sorted

Positional records are very compact if the defaults are fine

Questions?

Steve Love

@IAmSteveLove

steve@arventech.com