with expression - Nondestructive mutation creates a new object with modified properties

Article • 03/22/2023

Available in C# 9.0 and later, a `with` expression produces a copy of its operand with the specified properties and fields modified. You use the object initializer syntax to specify what members to modify and their new values:

C#

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1:
NamedPoint { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}");  // output: p2:
NamedPoint { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
            {
                Name = "C",
                Y = 4
            };
        Console.WriteLine($"{nameof(p3)}: {p3}");  // output: p3:
NamedPoint { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1:
NamedPoint { Name = A, X = 0, Y = 0 }

        var apples = new { Item = "Apples", Price = 1.19m };
        Console.WriteLine($"Original: {apples}");  // output:
Original: { Item = Apples, Price = 1.19 }
        var saleApples = apples with { Price = 0.79m };
        Console.WriteLine($"Sale: {saleApples}");  // output: Sale: {
Item = Apples, Price = 0.79 }
    }
}
```

In C# 9.0, a left-hand operand of a `with` expression must be of a record type. Beginning with C# 10, a left-hand operand of a `with` expression can also be of a structure type or an anonymous type.

The result of a `with` expression has the same run-time type as the expression's operand, as the following example shows:

C#

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X,
Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint);  // output: True
        Console.WriteLine(p2);  // output: NamedPoint { X = 5, Y = 3,
Name = A }
    }
}
```

In the case of a reference-type member, only the reference to a member instance is copied when an operand is copied. Both the copy and original operand have access to the same reference-type instance. The following example demonstrates that behavior:

C#

```
using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A",
"B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}:
```

```
{copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}:
{copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}
```

Custom copy semantics

Any record class type has the *copy constructor*. A *copy constructor* is a constructor with a single parameter of the containing record type. It copies the state of its argument to a new record instance. At evaluation of a `with` expression, the copy constructor gets called to instantiate a new record instance based on an original record. After that, the new instance gets updated according to the specified modifications. By default, the copy constructor is implicit, that is, compiler-generated. If you need to customize the record copy semantics, explicitly declare a copy constructor with the desired behavior. The following example updates the preceding example with an explicit copy constructor. The new copy behavior is to copy list items instead of a list reference when a record is copied:

C#

```
using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A",
"B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}:
```

```
{copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}:
{copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

You can't customize the copy semantics for structure types.

C# language specification

For more information, see the following sections of the records feature proposal note:

- with expression
- Copy and Clone members

See also

- C# reference
- C# operators and expressions
- Records
- Structure types