Welcome to C# 9.0

Mads

May 20th, 2020

Note: This post is out of date. Now that C# 9.0 has been released, an updated version can be found [here](#).

C# 9.0 is taking shape, and I'd like to share our thinking on some of the major features we're adding to this next version of the language.

With every new version of C# we strive for greater clarity and simplicity in common coding scenarios, and C# 9.0 is no exception. One particular focus this time is supporting terse and immutable representation of data shapes.

Let's dive in!

Init-only properties

Object initializers are pretty awesome. They give the client of a type a very flexible and readable format for creating an object, and they are especially great for nested object creation where a whole tree of objects is created in one go. Here's a simple one:

```
new Person
{
    FirstName = "Scott",
    LastName = "Hunter"
}
```

Object initializers also free the type author from writing a lot of construction boilerplate – all they have to do is write some properties!

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

The one big limitation today is that the properties have to be *mutable* for object initializers to work: They function by first calling the object's constructor (the default, parameterless one in this case) and then assigning to the property setters.

Init-only properties fix that! They introduce an `init` accessor that is a variant of the `set` accessor which can only be called during object initialization:

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

With this declaration, the client code above is still legal, but any subsequent assignment to the `FirstName` and `LastName` properties is an error.

Init accessors and readonly fields

Because `init` accessors can only be called during initialization, they are allowed to mutate `readonly` fields of the enclosing class, just like you can in a constructor.

```
public class Person
{
    private readonly string firstName;
    private readonly string lastName;

    public string FirstName
    {
        get => firstName;
        init => firstName = (value ?? throw new ArgumentNullException(nameof(FirstName)));
    }
    public string LastName
    {
        get => lastName;
        init => lastName = (value ?? throw new ArgumentNullException(nameof(LastName)));
    }
}
```

Records

Init-only properties are great if you want to make individual properties immutable. If you want the whole object to be immutable and behave like a value, then you should consider declaring it as a *record*:

```
public data class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

The `data` keyword on the class declaration marks it as a record. This imbues it with several additional value-like behaviors, which we'll dig into in the following. Generally speaking, records are meant to be seen more as "values" – data! – and less as objects. They aren't meant to have mutable encapsulated state. Instead you represent change over time by creating new records representing the new state. They are defined not by their identity, but by their contents.

With-expressions

When working with immutable data, a common pattern is to create new values from existing ones to represent a new state. For instance, if our person were to change their last name we would represent it as a new object that's a copy of the old one, except with a different last name. This technique is often referred to as *non-destructive mutation*. Instead of representing the person *over time*, the record represents the person's state *at a given time*.

To help with this style of programming, records allow for a new kind of expression; the `with`-expression:

```
var otherPerson = person with { LastName = "Hanselman" };
```

With-expressions use object initializer syntax to state what's different in the new object from the old object. You can specify multiple properties.

A record implicitly defines a `protected` "copy constructor" – a constructor that takes an existing record object and copies it field by field to the new one:

```
protected Person(Person original) { /* copy all the fields */ } // generated
```

The `with` expression causes the copy constructor to get called, and then applies the object initializer on top to change the properties accordingly.

If you don't like the default behavior of the generated copy constructor you can define your own instead, and that will be picked up by the `with` expression.

Value-based equality

All objects inherit a virtual `Equals(object)` method from the `object` class. This is used as the basis for the `Object.Equals(object, object)` static method when both parameters are non-null.

Structs override this to have "value-based equality", comparing each field of the struct by calling `Equals` on them recursively. Records do the same.

This means that in accordance with their "value-ness" two record objects can be equal to one another without being the *same* object. For instance if we modify the last name of the modified person back again:

```
var originalPerson = otherPerson with { LastName = "Hunter" };
```

We would now have `ReferenceEquals(person, originalPerson)` = false (they aren't the same object) but `Equals(person, originalPerson)` = true (they have the same value).

If you don't like the default field-by-field comparison behavior of the generated `Equals` override, you can write your own instead. You just need to be careful that you understand how value-based equality works in records, especially when inheritance is involved, which we'll come back to below.

Along with the value-based `Equals` there's also a value-based `GetHashCode()` override to go along with it.

Data members

Records are overwhelmingly intended to be immutable, with init-only public properties that can be non-destructively modified through `with`-expressions. In order to optimize for that common case, records change the defaults of what a simple member declaration of the form `string FirstName` means. Instead of an implicitly private field, as in other class and struct declarations, in records this is taken to be shorthand for a public, init-only auto-property! Thus, the declaration:

```
public data class Person { string FirstName; string LastName; }
```

Means exactly the same as the one we had before:

```
public data class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

We think this makes for beautiful and clear record declarations. If you really want a private field, you can just add the `private` modifier explicitly:

```
private string firstName;
```

Positional records

Sometimes it's useful to have a more positional approach to a record, where its contents are given via constructor arguments, and can be extracted with positional deconstruction.

It's perfectly possible to specify your own constructor and deconstructor in a record:

```
public data class Person
{
    string FirstName;
    string LastName;
    public Person(string firstName, string lastName)
      => (FirstName, LastName) = (firstName, lastName);
    public void Deconstruct(out string firstName, out string lastName)
      => (firstName, lastName) = (FirstName, LastName);
}
```

But there's a much shorter syntax for expressing exactly the same thing (modulo casing of parameter names):

```
public data class Person(string FirstName, string LastName);
```

This declares the public init-only auto-properties *and* the constructor *and* the deconstructor, so that you can write:

```
var person = new Person("Scott", "Hunter"); // positional construction
var (f, l) = person;                        // positional deconstruction
```

If you don't like the generated auto-property you can define your own property of the same name instead, and the generated constructor and deconstructor will just use that one.

Records and mutation

The value-based semantics of a record don't gel well with mutable state. Imagine putting a record object into a dictionary. Finding it again depends on `Equals` and (sometimes) `GethashCode`. But if the record changes its state, it will also change what it's equal to! We might not be able to find it again! In a hash table implementation it might even corrupt the data structure, since placement is based on the hash code it has "on arrival"!

There are probably some valid advanced uses of mutable state inside of records, notably for caching. But the manual work involved in overriding the default behaviors to ignore such state is likely to be considerable.

With-expressions and inheritance

Value-based equality and non-destructive mutation are notoriously challenging when combined with inheritance. Let's add a derived record class `Student` to our running example:

```
public data class Person { string FirstName; string LastName; }
public data class Student : Person { int ID; }
```

And let's start our `with`-expression example by actually creating a `Student`, but storing it in a `Person` variable:

```
Person person = new Student { FirstName = "Scott", LastName = "Hunter", ID = GetNewId() };
otherPerson = person with { LastName = "Hanselman" };
```

At the point of that `with`-expression on the last line the compiler has no idea that `person` actually contains a `Student`. Yet, the new person wouldn't be a proper copy if it wasn't *actually* a `Student` object, complete with the same `ID` as the first one copied over.

C# makes this work. Records have a hidden virtual method that is entrusted with "cloning" the *whole* object. Every derived record type overrides this method to call the copy constructor of that type, and the copy constructor of a derived record chains to the copy constructor of the base record. A `with`-expression simply calls the hidden "clone" method and applies the object initializer to the result.

Value-based equality and inheritance

Similarly to the `with`-expression support, value-based equality also has to be "virtual", in the sense that `Student`s need to compare all the `Student` fields, even if the statically known type at the point of comparison is a base type like `Person`. That is easily achieved by overriding the already virtual `Equals` method.

However, there is an additional challenge with equality: What if you compare two *different* kinds of `Person`? We can't really just let one of them decide which equality to apply: Equality is supposed to be symmetric, so the result should be the same regardless of which of the two objects come first. In other words, they have to *agree* on the equality being applied!

An example to illustrate the problem:

```
Person person1 = new Person { FirstName = "Scott", LastName = "Hunter" };
Person person2 = new Student { FirstName = "Scott", LastName = "Hunter", ID = GetNewId() };
```

Are the two objects equal to one another? `person1` might think so, since `person2` has all the `Person` things right, but `person2` would beg to differ! We need to make sure that they both agree that they are different objects.

Once again, C# takes care of this for you automatically. The way it's done is that records have a virtual protected property called `EqualityContract`. Every derived record overrides it, and in order to compare equal, the two objects musts have the same `EqualityContract`.

Top-level programs

Writing a simple program in C# requires a remarkable amount of boilerplate code:

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello World!");
    }
}
```

This is not only overwhelming for language beginners, but clutters up the code and adds levels of indentation.

In C# 9.0 you can just choose to write your main program at the top level instead:

```
using System;

Console.WriteLine("Hello World!");
```

Any statement is allowed. The program has to occur after the `using`s and before any type or namespace declarations in the file, and you can only do this in one file, just as you can have only one `Main` method today.

If you want to return a status code you can do that. If you want to `await` things you can do that. And if you want to access command line arguments, `args` is available as a "magic" parameter.

Local functions are a form of statement and are also allowed in the top level program. It is an error to call them from anywhere outside of the top level statement section.

Improved pattern matching

Several new kinds of patterns have been added in C# 9.0. Let's look at them in the context of this code snippet from the [pattern matching tutorial](#):

```
public static decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
       ...

        DeliveryTruck t when t.GrossWeightClass > 5000 => 10.00m + 5.00m,
        DeliveryTruck t when t.GrossWeightClass < 3000 => 10.00m - 2.00m,
        DeliveryTruck _ => 10.00m,

        _ => throw new ArgumentException("Not a known vehicle type", nameof(vehicle))
    };
```

Simple type patterns

Currently, a type pattern needs to declare an identifier when the type matches – even if that identifier is a discard `_`, as in `DeliveryTruck _` above. But now you can just write the type:

```
DeliveryTruck => 10.00m,
```

Relational patterns

C# 9.0 introduces patterns corresponding to the relational operators `<`, `<=` and so on. So you can now write the `DeliveryTruck` part of the above pattern as a nested switch expression:

```
DeliveryTruck t when t.GrossWeightClass switch
{
    > 5000 => 10.00m + 5.00m,
    < 3000 => 10.00m - 2.00m,
    _ => 10.00m,
},
```

Here `> 5000` and `< 3000` are relational patterns.

Logical patterns

Finally you can combine patterns with logical operators `and`, `or` and `not`, spelled out as words to avoid confusion with the operators used in expressions. For instance, the cases of the nested switch above could be put into ascending order like this:

```
DeliveryTruck t when t.GrossWeightClass switch
{
    < 3000 => 10.00m - 2.00m,
    >= 3000 and <= 5000 => 10.00m,
    > 5000 => 10.00m + 5.00m,
},
```

The middle case there uses `and` to combine two relational patterns and form a pattern representing an interval.

A common use of the `not` pattern will be applying it to the `null` constant pattern, as in `not null`. For instance we can split the handling of unknown cases depending on whether they are null:

```
not null => throw new ArgumentException($"Not a known vehicle type: {vehicle}", nameof(vehic
null => throw new ArgumentNullException(nameof(vehicle))
```

Also `not` is going to be convenient in if-conditions containing is-expressions where, instead of unwieldy double parentheses:

```
if (!(e is Customer)) { ... }
```

You can just say

```
if (e is not Customer) { ... }
```

Improved target typing

"Target typing" is a term we use for when an expression gets its type from the context of where it's being used. For instance `null` and lambda expressions are always target typed.

In C# 9.0 some expressions that weren't previously target typed become able to be guided by their context.

Target-typed `new` expressions

`new` expressions in C# have always required a type to be specified (except for implicitly typed array expressions). Now you can leave out the type if there's a clear type that the expressions is being assigned to.

```
Point p = new (3, 5);
```

Target typed `??` and `?:`

Sometimes conditional `??` and `?:` expressions don't have an obvious shared type between the branches. Such cases fail today, but C# 9.0 will allow them if there's a target type that both branches convert to:

```
Person person = student ?? customer; // Shared base type
int? result = b ? 0 : null; // nullable value type
```

Covariant returns

It's sometimes useful to express that a method override in a derived class has a more specific return type than the declaration in the base type. C# 9.0 allows that:

```
abstract class Animal
{
    public abstract Food GetFood();
    ...
}
class Tiger : Animal
{
    public override Meat GetFood() => ...;
}
```