Create record types

Article • 11/22/2024

Records are types that use *value-based equality*. You can define records as reference types or value types. Two variables of a record type are equal if the record type definitions are identical, and if for every field, the values in both records are equal. Two variables of a class type are equal if the objects referred to are the same class type and the variables refer to the same object. Value-based equality implies other capabilities you probably want in record types. The compiler generates many of those members when you declare a `record` instead of a `class`. The compiler generates those same methods for `record struct` types.

In this tutorial, you learn how to:

- ✓ Decide if you add the `record` modifier to a `class` type.
- ✓ Declare record types and positional record types.
- ✓ Substitute your methods for compiler generated methods in records.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Characteristics of records

You define a *record* by declaring a type with the `record` keyword, modifying a `class` or `struct` declaration. Optionally, you can omit the `class` keyword to create a `record class`. A record follows value-based equality semantics. To enforce value semantics, the compiler generates several methods for your record type (both for `record class` types and `record struct` types):

- An override of Object.Equals(Object).
- A virtual `Equals` method whose parameter is the record type.
- An override of Object.GetHashCode().
- Methods for `operator ==` and `operator !=`.
- Record types implement System.IEquatable<T>.

Records also provide an override of Object.ToString(). The compiler synthesizes methods for displaying records using Object.ToString(). You explore those members as you write the code for this tutorial. Records support `with` expressions to enable nondestructive mutation of records.

You can also declare *positional records* using a more concise syntax. The compiler synthesizes more methods for you when you declare positional records:

- A primary constructor whose parameters match the positional parameters on the record declaration.
- Public properties for each parameter of a primary constructor. These properties are *init-only* for `record class` types and `readonly record struct` types. For `record struct` types, they're *read-write*.
- A `Deconstruct` method to extract properties from the record.

Build temperature data

Data and statistics are among the scenarios where you want to use records. For this tutorial, you build an application that computes *degree days* for different uses. *Degree days* are a measure of heat (or lack of heat) over a period of days, weeks, or months. Degree days track and predict energy usage. More hotter days mean more air conditioning, and more colder days mean more furnace usage. Degree days help manage plant populations and correlate to plant growth as the seasons change. Degree days help track animal migrations for species that travel to match climate.

The formula is based on the mean temperature on a given day and a baseline temperature. To compute degree days over time, you'll need the high and low temperature each day for a period of time. Let's start by creating a new application. Make a new console application. Create a new record type in a new file named "DailyTemperature.cs":

```C#
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp);
```

The preceding code defines a *positional record*. The `DailyTemperature` record is a `readonly record struct`, because you don't intend to inherit from it, and it should be immutable. The `HighTemp` and `LowTemp` properties are *init only properties*, meaning they can be set in the constructor or using a property initializer. If you wanted the positional parameters to be read-write, you declare a `record struct` instead of a `readonly record struct`. The `DailyTemperature` type also has a *primary constructor* that has two parameters that match the two properties. You use the primary constructor to initialize a

`DailyTemperature` record. The following code creates and initializes several `DailyTemperature` records. The first uses named parameters to clarify the `HighTemp` and `LowTemp`. The remaining initializers use positional parameters to initialize the `HighTemp` and `LowTemp`:

C#

```
private static DailyTemperature[] data = [
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
];
```

You can add your own properties or methods to records, including positional records. You need to compute the mean temperature for each day. You can add that property to the `DailyTemperature` record:

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Let's make sure you can use this data. Add the following code to your `Main` method:

C#

```
foreach (var item in data)
    Console.WriteLine(item);
```

Run your application, and you see output that looks similar to the following display (several rows removed for space):

.NET CLI

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }


DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

The preceding code shows the output from the override of `ToString` synthesized by the compiler. If you prefer different text, you can write your own version of `ToString` that prevents the compiler from synthesizing a version for you.

Compute degree days

To compute degree days, you take the difference from a baseline temperature and the mean temperature on a given day. To measure heat over time, you discard any days where the mean temperature is below the baseline. To measure cold over time, you discard any days where the mean temperature is above the baseline. For example, the U.S. uses 65 F as the base for both heating and cooling degree days. That's the temperature where no heating or cooling is needed. If a day has a mean temperature of 70 F, that day is five cooling degree days and zero heating degree days. Conversely, if the mean temperature is 55 F, that day is 10 heating degree days and 0 cooling degree days.

You can express these formulas as a small hierarchy of record types: an abstract degree day type and two concrete types for heating degree days and cooling degree days. These types can also be positional records. They take a baseline temperature and a sequence of daily temperature records as arguments to the primary constructor:

C#

```csharp
public abstract record DegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean <
BaseTemperature).Sum(s => BaseTemperature - s.Mean);
}
```

```
public sealed record CoolingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean >
BaseTemperature).Sum(s => s.Mean - BaseTemperature);
}
```

The abstract `DegreeDays` record is the shared base class for both the `HeatingDegreeDays` and `CoolingDegreeDays` records. The primary constructor declarations on the derived records show how to manage base record initialization. Your derived record declares parameters for all the parameters in the base record primary constructor. The base record declares and initializes those properties. The derived record doesn't hide them, but only creates and initializes properties for parameters that aren't declared in its base record. In this example, the derived records don't add new primary constructor parameters. Test your code by adding the following code to your `Main` method:

C#

```csharp
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

You get output like the following display:

.NET CLI

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 71.5 }
```

Define compiler-synthesized methods

Your code calculates the correct number of heating and cooling degree days over that period of time. But this example shows why you might want to replace some of the synthesized methods for records. You can declare your own version of any of the compiler-synthesized methods in a record type except the clone method. The clone method has a compiler-generated name and you can't provide a different implementation. These synthesized methods include a copy constructor, the members of the System.IEquatable<T> interface, equality and inequality tests, and GetHashCode().

For this purpose, you synthesize `PrintMembers`. You could also declare your own `ToString`, but `PrintMembers` provides a better option for inheritance scenarios. To provide your own version of a synthesized method, the signature must match the synthesized method.

The `TempRecords` element in the console output isn't useful. It displays the type, but nothing else. You can change this behavior by providing your own implementation of the synthesized `PrintMembers` method. The signature depends on modifiers applied to the `record` declaration:

- If a record type is `sealed`, or a `record struct`, the signature is `private bool PrintMembers(StringBuilder builder);`
- If a record type isn't `sealed` and derives from `object` (that is, it doesn't declare a base record), the signature is `protected virtual bool PrintMembers(StringBuilder builder);`
- If a record type isn't `sealed` and derives from another record, the signature is `protected override bool PrintMembers(StringBuilder builder);`

These rules are easiest to comprehend through understanding the purpose of `PrintMembers`. `PrintMembers` adds information about each property in a record type to a string. The contract requires base records to add their members to the display and assumes derived members add their members. Each record type synthesizes a `ToString` override that looks similar to the following example for `HeatingDegreeDays`:

```C#
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

You declare a `PrintMembers` method in the `DegreeDays` record that doesn't print the type of the collection:

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

The signature declares a `virtual protected` method to match the compiler's version. Don't worry if you get the accessors wrong; the language enforces the correct signature. If you forget the correct modifiers for any synthesized method, the compiler issues warnings or errors that help you get the right signature.

You can declare the `ToString` method as `sealed` in a record type. That prevents derived records from providing a new implementation. Derived records will still contain the `PrintMembers` override. You would seal `ToString` if you didn't want it to display the runtime type of the record. In the preceding example, you'd lose the information on where the record was measuring heating or cooling degree days.

Nondestructive mutation

The synthesized members in a positional record class don't modify the state of the record. The goal is that you can more easily create immutable records. Remember that you declare a `readonly record struct` to create an immutable record struct. Look again at the preceding declarations for `HeatingDegreeDays` and `CoolingDegreeDays`. The members added perform computations on the values for the record, but don't mutate state. Positional records make it easier for you to create immutable reference types.

Creating immutable reference types means you want to use nondestructive mutation. You create new record instances that are similar to existing record instances using with expressions. These expressions are a copy construction with extra assignments that modify the copy. The result is a new record instance where each property was copied from the existing record and optionally modified. The original record is unchanged.

Let's add a couple features to your program that demonstrate `with` expressions. First, let's create a new record to compute growing degree days using the same data. *Growing degree days* typically uses 41 F as the baseline and measures temperatures above the baseline. To use the same data, you can create a new record that is similar to the `coolingDegreeDays`, but with a different base temperature:

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
```

```
Console.WriteLine(growingDegreeDays);
```

You can compare the number of degrees computed to the numbers generated with a higher baseline temperature. Remember that records are *reference types* and these copies are shallow copies. The array for the data isn't copied, but both records refer to the same data. That fact is an advantage in one other scenario. For growing degree days, it's useful to keep track of the total for the previous five days. You can create new records with different source data using `with` expressions. The following code builds a collection of these accumulations, then displays the values:

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..
(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

You can also use `with` expressions to create copies of records. Don't specify any properties between the braces for the `with` expression. That means create a copy, and don't change any properties:

C#

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Run the finished application to see the results.

Summary

This tutorial showed several aspects of records. Records provide concise syntax for types where the fundamental use is storing data. For object-oriented classes, the fundamental use is defining responsibilities. This tutorial focused on *positional records*, where you can use a concise syntax to declare the properties for a record. The compiler synthesizes several members of the record for copying and comparing records. You can add any other members you need for your record types. You can create immutable record types knowing that none of the compiler-generated members would mutate state. And `with` expressions make it easy to support nondestructive mutation.

Records add another way to define types. You use `class` definitions to create object-oriented hierarchies that focus on the responsibilities and behavior of objects. You create `struct` types for data structures that store data and are small enough to copy efficiently. You create `record` types when you want value-based equality and comparison, don't want to copy values, and want to use reference variables. You create `record struct` types when you want the features of records for a type that is small enough to copy efficiently.

You can learn more about records in the C# language reference article for the record type and the proposed record type specification and record struct specification.