Records (C# reference)

Article • 05/25/2023

Beginning with C# 9, you use the `record` modifier to define a reference type that provides built-in functionality for encapsulating data. C# 10 allows the `record class` syntax as a synonym to clarify a reference type, and `record struct` to define a value type with similar functionality.

When you declare a primary constructor on a record, the compiler generates public properties for the primary constructor parameters. The primary constructor parameters to a record are referred to as *positional parameters*. The compiler creates *positional properties* that mirror the primary constructor or positional parameters. The compiler doesn't synthesize properties for primary constructor parameters on types that don't have the `record` modifier.

The following two examples demonstrate `record` (or `record class`) reference types:

```csharp
public record Person(string FirstName, string LastName);
```

```csharp
public record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
};
```

The following two examples demonstrate `record struct` value types:

```csharp
public readonly record struct Point(double X, double Y, double Z);
```

```csharp
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
    public double Z { get; init; }
}
```

You can also create records with mutable properties and fields:

C#

```csharp
public record Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
};
```

Record structs can be mutable as well, both positional record structs and record structs with no positional parameters:

C#

```csharp
public record struct DataMeasurement(DateTime TakenAt, double
Measurement);
```

C#

```csharp
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

While records can be mutable, they're primarily intended for supporting immutable data models. The record type offers the following features:

- Concise syntax for creating a reference type with immutable properties
- Built-in behavior useful for a data-centric reference type:
  - Value equality
  - Concise syntax for nondestructive mutation
  - Built-in formatting for display
- Support for inheritance hierarchies

The preceding examples show some distinctions between records that are reference types and records that are value types:

- A `record` or a `record class` declares a reference type. The `class` keyword is optional, but can add clarity for readers. A `record struct` declares a value type.
- Positional properties are *immutable* in a `record class` and a `readonly record struct`. They're *mutable* in a `record struct`.

The remainder of this article discusses both `record class` and `record struct` types. The differences are detailed in each section. You should decide between a `record class` and a `record struct` similar to deciding between a `class` and a `struct`. The term *record* is used to describe behavior that applies to all record types. Either `record struct` or `record class` is used to describe behavior that applies to only struct or class types, respectively. The `record` type was introduced in C# 9; `record struct` types were introduced in C# 10.

Positional syntax for property definition

You can use positional parameters to declare properties of a record and to initialize the property values when you create an instance:

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

When you use the positional syntax for property definition, the compiler creates:

- A public autoimplemented property for each positional parameter provided in the record declaration.
  - For `record` types and `readonly record struct` types: An init-only property.
  - For `record struct` types: A read-write property.
- A primary constructor whose parameters match the positional parameters on the record declaration.
- For record struct types, a parameterless constructor that sets each field to its default value.
- A `Deconstruct` method with an `out` parameter for each positional parameter provided in the record declaration. The method deconstructs properties defined by using positional syntax; it ignores properties that are defined by using standard property syntax.

You may want to add attributes to any of these elements the compiler creates from the record definition. You can add a *target* to any attribute you apply to the positional record's properties. The following example applies the

System.Text.Json.Serialization.JsonPropertyNameAttribute to each property of the `Person` record. The `property:` target indicates that the attribute is applied to the compiler-generated property. Other values are `field:` to apply the attribute to the field, and `param:` to apply the attribute to the parameter.

C#

```
/// <summary>
/// Person record type
/// </summary>
/// <param name="FirstName">First Name</param>
/// <param name="LastName">Last Name</param>
/// <remarks>
/// The person type is a positional record containing the
/// properties for the first and last name. Those properties
/// map to the JSON elements "firstName" and "lastName" when
/// serialized or deserialized.
/// </remarks>
public record Person([property: JsonPropertyName("firstName")]
string FirstName,
    [property: JsonPropertyName("lastName")] string LastName);
```

The preceding example also shows how to create XML documentation comments for the record. You can add the `<param>` tag to add documentation for the primary constructor's parameters.

If the generated autoimplemented property definition isn't what you want, you can define your own property of the same name. For example, you may want to change accessibility or mutability, or provide an implementation for either the `get` or `set` accessor. If you declare the property in your source, you must initialize it from the positional parameter of the record. If your property is an autoimplemented property, you must initialize the property. If you add a backing field in your source, you must initialize the backing field. The generated deconstructor uses your property definition. For instance, the following example declares the `FirstName` and `LastName` properties of a positional record `public`, but restricts the `Id` positional parameter to `internal`. You can use this syntax for records and record struct types.

C#

```
public record Person(string FirstName, string LastName, string Id)
{
    internal string Id { get; init; } = Id;
}

public static void Main()
{
    Person person = new("Nancy", "Davolio", "12345");
    Console.WriteLine(person.FirstName); //output: Nancy

}
```

A record type doesn't have to declare any positional properties. You can declare a record without any positional properties, and you can declare other fields and properties, as in the following example:

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; } = Array.Empty<string>
();
};
```

If you define properties by using standard property syntax but omit the access modifier, the properties are implicitly `private`.

Immutability

A *positional record* and a *positional readonly record struct* declare init-only properties. A *positional record struct* declares read-write properties. You can override either of those defaults, as shown in the previous section.

Immutability can be useful when you need a data-centric type to be thread-safe or you're depending on a hash code remaining the same in a hash table. Immutability isn't appropriate for all data scenarios, however. Entity Framework Core, for example, doesn't support updating with immutable entity types.

Init-only properties, whether created from positional parameters (`record class`, and `readonly record struct`) or by specifying `init` accessors, have *shallow immutability*. After initialization, you can't change the value of value-type properties or the reference of reference-type properties. However, the data that a reference-type property refers to can be changed. The following example shows that the content of a reference-type immutable property (an array in this case) is mutable:

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-
1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

The features unique to record types are implemented by compiler-synthesized methods, and none of these methods compromises immutability by modifying object state. Unless specified, the synthesized methods are generated for `record`, `record struct`, and `readonly record struct` declarations.

Value equality

If you don't override or replace equality methods, the type you declare governs how equality is defined:

- For `class` types, two objects are equal if they refer to the same object in memory.
- For `struct` types, two objects are equal if they are of the same type and store the same values.
- For types with the `record` modifier (`record class`, `record struct`, and `readonly record struct`), two objects are equal if they are of the same type and store the same values.

The definition of equality for a `record struct` is the same as for a `struct`. The difference is that for a `struct`, the implementation is in ValueType.Equals(Object) and relies on reflection. For records, the implementation is compiler synthesized and uses the declared data members.

Reference equality is required for some data models. For example, Entity Framework Core depends on reference equality to ensure that it uses only one instance of an entity type for what is conceptually one entity. For this reason, records and record structs aren't appropriate for use as entity types in Entity Framework Core.

The following example illustrates value equality of record types:

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output:
False
}
```

To implement value equality, the compiler synthesizes several methods, including:

- An override of Object.Equals(Object). It's an error if the override is declared explicitly.

  This method is used as the basis for the Object.Equals(Object, Object) static method when both parameters are non-null.

- A `virtual`, or `sealed`, `Equals(R? other)` where `R` is the record type. This method implements IEquatable<T>. This method can be declared explicitly.

- If the record type is derived from a base record type `Base`, `Equals(Base? other)`. It's an error if the override is declared explicitly. If you provide your own implementation of `Equals(R? other)`, provide an implementation of `GetHashCode` also.

- An override of Object.GetHashCode(). This method can be declared explicitly.

- Overrides of operators `==` and `!=`. It's an error if the operators are declared explicitly.

- If the record type is derived from a base record type, `protected override Type EqualityContract { get; };`. This property can be declared explicitly. For more information, see Equality in inheritance hierarchies.

The compiler doesn't synthesize a method when a record type has a method that matches the signature of a synthesized method allowed to be declared explicitly.

Nondestructive mutation

If you need to copy an instance with some modifications, you can use a `with` expression to achieve *nondestructive mutation*. A `with` expression makes a new record instance that is a copy of an existing record instance, with specified properties and fields modified. You use object initializer syntax to specify the values to be changed, as shown in the following example:

```C#
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new
string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio,
PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio,
PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio,
PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

The `with` expression can set positional properties or properties created by using standard property syntax. Explicitly declared properties must have an `init` or `set` accessor to be changed in a `with` expression.

The result of a `with` expression is a *shallow copy*, which means that for a reference property, only the reference to an instance is copied. Both the original record and the copy end up with a reference to the same instance.

To implement this feature for `record class` types, the compiler synthesizes a clone method and a copy constructor. The virtual clone method returns a new record initialized by the copy constructor. When you use a `with` expression, the compiler creates code that calls the clone method and then sets the properties that are specified in the `with` expression.

If you need different copying behavior, you can write your own copy constructor in a `record class`. If you do that, the compiler doesn't synthesize one. Make your constructor `private` if the record is `sealed`, otherwise make it `protected`. The compiler doesn't synthesize a copy constructor for `record struct` types. You can write one, but the compiler doesn't generate calls to it for `with` expressions. The values of the `record struct` are copied on assignment.

You can't override the clone method, and you can't create a member named `Clone` in any record type. The actual name of the clone method is compiler-generated.

Built-in formatting for display

Record types have a compiler-generated ToString method that displays the names and values of public properties and fields. The `ToString` method returns a string of the following format:

> <record type name> { <property name> = <value>, <property name> = <value>, …}

The string printed for `<value>` is the string returned by the ToString() for the type of the property. In the following example, `ChildNames` is a System.Array, where `ToString` returns `System.String[]`:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames =
System.String[] }
```

To implement this feature, in `record class` types, the compiler synthesizes a virtual `PrintMembers` method and a ToString override. In `record struct` types, this member is `private`. The `ToString` override creates a StringBuilder object with the type name followed by an opening bracket. It calls `PrintMembers` to add property names and values, then adds the closing bracket. The following example shows code similar to what the synthesized override contains:

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

You can provide your own implementation of `PrintMembers` or the `ToString` override. Examples are provided in the PrintMembers formatting in derived records section later in this article. In C# 10 and later, your implementation of `ToString` may include the `sealed` modifier, which prevents the compiler from synthesizing a `ToString` implementation for any derived records. You can create a consistent string representation throughout a hierarchy of `record` types. (Derived records still have a `PrintMembers` method generated for all derived properties.)

Inheritance

This section only applies to `record class` types.

A record can inherit from another record. However, a record can't inherit from a class, and a class can't inherit from a record.

Positional parameters in derived record types

The derived record declares positional 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.

The following example illustrates inheritance with positional property syntax:

```C#
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade
= 3 }
}
```

Equality in inheritance hierarchies

This section applies to `record class` types, but not `record struct` types. For two record variables to be equal, the run-time type must be equal. The types of the containing variables might be different. Inherited equality comparison is illustrated in the following code example:

C#

```csharp
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

In the example, all variables are declared as `Person`, even when the instance is a derived type of either `Student` or `Teacher`. The instances have the same properties and the same property values. But `student == teacher` returns `False` although both are `Person`-type variables, and `student == student2` returns `True` although one is a `Person` variable and one is a `Student` variable. The equality test depends on the runtime type of the actual object, not the declared type of the variable.

To implement this behavior, the compiler synthesizes an `EqualityContract` property that returns a Type object that matches the type of the record. The `EqualityContract` enables the equality methods to compare the runtime type of objects when they're checking for equality. If the base type of a record is `object`, this property is `virtual`. If the base type is another record type, this property is an override. If the record type is `sealed`, this property is effectively `sealed` because the type is `sealed`.

When code compares two instances of a derived type, the synthesized equality methods check all data members of the base and derived types for equality. The synthesized `GetHashCode` method uses the `GetHashCode` method from all data members declared in the base type and the derived record type. The data members of a `record` include all declared fields and the compiler-synthesized backing field for any autoimplemented properties.

`with` expressions in derived records

The result of a `with` expression has the same run-time type as the expression's operand. All properties of the run-time type get copied, but you can only set properties of the compile-time type, as the following example shows:

C#

```csharp
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name
or Zderived
    Console.WriteLine(p2 is NamedPoint);  // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A,
Zderived = 4 }

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase
= 7, Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B,
Zderived = 8 }
}
```

`PrintMembers` formatting in derived records

The synthesized `PrintMembers` method of a derived record type calls the base implementation. The result is that all public properties and fields of both derived and base types are included in the `ToString` output, as shown in the following example:

C#

```csharp
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
```

```
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade
= 3 }
}
```

You can provide your own implementation of the `PrintMembers` method. If you do that, use the following signature:

- For a `sealed` record that derives from `object` (doesn't declare a base record): `private bool PrintMembers(StringBuilder builder)`;
- For a `sealed` record that derives from another record (note that the enclosing type is `sealed`, so the method is effectively `sealed`): `protected override bool PrintMembers(StringBuilder builder)`;
- For a record that isn't `sealed` and derives from object: `protected virtual bool PrintMembers(StringBuilder builder);`
- For a record that isn't `sealed` and derives from another record: `protected override bool PrintMembers(StringBuilder builder);`

Here's an example of code that replaces the synthesized `PrintMembers` methods, one for a record type that derives from object, and one for a record type that derives from another record:

C#

```
public abstract record Person(string FirstName, string LastName,
string[] PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName =
{LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]},
PhoneNumber2 = {PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[]
PhoneNumbers, int Grade)
    : Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
            stringBuilder.Append(", ");
        };
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] {
"555-1234", "555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio,
PhoneNumber1 = 555-1234, PhoneNumber2 = 555-6789, Grade = 3 }
}
```

ⓘ **Note**

In C# 10 and later, the compiler will synthesize `PrintMembers` in derived records even when a base record has sealed the `ToString` method. You can also create your own implementation of `PrintMembers`.

Deconstructor behavior in derived records

The `Deconstruct` method of a derived record returns the values of all positional properties of the compile-time type. If the variable type is a base record, only the base record properties are deconstructed unless the object is cast to the derived type. The following example demonstrates calling a deconstructor on a derived record.

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy,
Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy,
Davolio, 3
}
```

Generic constraints

The `record` keyword is a modifier for either a `class` or `struct` type. Adding the `record` modifier includes the behavior described earlier in this article. There's no generic constraint that requires a type to be a record. A `record class` satisfies the `class` constraint. A `record struct` satisfies the `struct` constraint. For more information, see Constraints on type parameters.

C# language specification

For more information, see the Classes section of the C# language specification.

For more information about features introduced in C# 9 and later, see the following feature proposal notes:

- Records
- Init-only setters
- Covariant returns

See also

- C# reference
- Design guidelines - Choosing between class and struct
- Design guidelines - Struct design
- The C# type system
- with expression