Properties

Article • 09/29/2022 • 10 minutes to read

Properties are first class citizens in C#. The language defines syntax that enables developers to write code that accurately expresses their design intent.

Properties behave like fields when they're accessed. However, unlike fields, properties are implemented with accessors that define the statements executed when a property is accessed or assigned.

Property syntax

The syntax for properties is a natural extension to fields. A field defines a storage location:

C#

```
public class Person
{
    public string FirstName;

    // Omitted for brevity.
}
```

A property definition contains declarations for a `get` and `set` accessor that retrieves and assigns the value of that property:

C#

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

    // Omitted for brevity.
}
```

The syntax shown above is the *auto property* syntax. The compiler generates the storage location for the field that backs up the property. The compiler also implements the body of the `get` and `set` accessors.

Sometimes, you need to initialize a property to a value other than the default for its type. C# enables that by setting a value after the closing brace for the property. You may prefer the initial value for the `FirstName` property to be the empty string rather than `null`. You would specify that as shown below:

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // Omitted for brevity.
}
```

Specific initialization is most useful for read-only properties, as you'll see later in this article.

You can also define the storage yourself, as shown below:

C#

```
public class Person
{
    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }
    private string _firstName;

    // Omitted for brevity.
}
```

When a property implementation is a single expression, you can use *expression-bodied members* for the getter or setter:

C#

```
public class Person
{
    public string FirstName
    {
        get => _firstName;
        set => _firstName = value;
```

```
    }
    private string _firstName;

    // Omitted for brevity.
}
```

This simplified syntax will be used where applicable throughout this article.

The property definition shown above is a read-write property. Notice the keyword `value` in the set accessor. The `set` accessor always has a single parameter named `value`. The `get` accessor must return a value that is convertible to the type of the property (`string` in this example).

That's the basics of the syntax. There are many different variations that support various different design idioms. Let's explore, and learn the syntax options for each.

Validation

The examples above showed one of the simplest cases of property definition: a read-write property with no validation. By writing the code you want in the `get` and `set` accessors, you can create many different scenarios.

You can write code in the `set` accessor to ensure that the values represented by a property are always valid. For example, suppose one rule for the `Person` class is that the name can't be blank or white space. You would write that as follows:

C#

```
public class Person
{
    public string FirstName
    {
        get => _firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            _firstName = value;
        }
    }
    private string _firstName;

    // Omitted for brevity.
```

```
}
```

The preceding example can be simplified by using a `throw` expression as part of the property setter validation:

C#

```
public class Person
{
    public string FirstName
    {
        get => _firstName;
        set => _firstName = (!string.IsNullOrWhiteSpace(value)) ? value :
throw new ArgumentException("First name must not be blank");
    }
    private string _firstName;

    // Omitted for brevity.
}
```

The example above enforces the rule that the first name must not be blank or white space. If a developer writes

C#

```
hero.FirstName = "";
```

That assignment throws an `ArgumentException`. Because a property set accessor must have a void return type, you report errors in the set accessor by throwing an exception.

You can extend this same syntax to anything needed in your scenario. You can check the relationships between different properties, or validate against any external conditions. Any valid C# statements are valid in a property accessor.

Access control

Up to this point, all the property definitions you have seen are read/write properties with public accessors. That's not the only valid accessibility for properties. You can create read-only properties, or give different accessibility to the set and get accessors. Suppose that your `Person` class should only enable changing the value of the `FirstName` property from other methods in that class. You could give the set accessor `private` accessibility instead of `public`:

C#

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

    // Omitted for brevity.
}
```

Now, the `FirstName` property can be accessed from any code, but it can only be assigned from other code in the `Person` class.

You can add any restrictive access modifier to either the set or get accessors. Any access modifier you place on the individual accessor must be more limited than the access modifier on the property definition. The above is legal because the `FirstName` property is `public`, but the set accessor is `private`. You couldn't declare a `private` property with a `public` accessor. Property declarations can also be declared `protected`, `internal`, `protected internal`, or, even `private`.

It's also legal to place the more restrictive modifier on the `get` accessor. For example, you could have a `public` property, but restrict the `get` accessor to `private`. That scenario is rarely done in practice.

Read-only

You can also restrict modifications to a property so that it can only be set in a constructor. You can modify the `Person` class so as follows:

C#

```
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

Init-only

The preceding example requires callers to use the constructor that includes the `FirstName` parameter. Callers can't use object initializers to assign a value to the property. To support initializers, you can make the `set` accessor an `init` accessor, as shown in the following code:

C#

```
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; init; }

    // Omitted for brevity.
}
```

The preceding example allows a caller to create a `Person` using the default constructor, even when that code doesn't set the `FirstName` property. Beginning in C# 11, you can *require* callers to set that property:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }
```

```
    // Omitted for brevity.
}
```

The preceding code makes two addition to the `Person` class. First, the `FirstName` property declaration includes the `required` modifier. That means any code that creates a new `Person` must set this property. Second, the constructor that takes a `firstName` parameter has the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute attribute. This attribute informs the compiler that this constructor sets *all* `required` members.

> ⓘ **Important**
>
> Don't confuse `required` with *non-nullable*. It's valid to set a `required` property to `null` or `default`. If the type is non-nullable, such as `string` in these examples, the compiler issues a warning.

Callers must either use the constructor with `SetsRequiredMembers` or set the `FirstName` property using an object initializer, as shown in the following code:

C#

```csharp
var person = new VersionNinePoint2.Person("John");
person = new VersionNinePoint2.Person{ FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//person = new VersionNinePoint2.Person();
```

Computed properties

A property doesn't need to simply return the value of a member field. You can create properties that return a computed value. Let's expand the `Person` object to return the full name, computed by concatenating the first and last names:

C#

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

```
    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

The example above uses the string interpolation feature to create the formatted string for the full name.

You can also use an *expression-bodied member*, which provides a more succinct way to create the computed `FullName` property:

C#

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

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Expression-bodied members use the *lambda expression* syntax to define methods that contain a single expression. Here, that expression returns the full name for the person object.

Cached evaluated properties

You can mix the concept of a computed property with storage and create a *cached evaluated property*. For example, you could update the `FullName` property so that the string formatting only happened the first time it was accessed:

C#

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

    public string LastName { get; set; }

    private string _fullName;
    public string FullName
    {
```

```
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

The above code contains a bug though. If code updates the value of either the `FirstName` or `LastName` property, the previously evaluated `fullName` field is invalid. You modify the `set` accessors of the `FirstName` and `LastName` property so that the `fullName` field is calculated again:

C#

```csharp
public class Person
{
    private string _firstName;
    public string FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            _fullName = null;
        }
    }

    private string _lastName;
    public string LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            _fullName = null;
        }
    }

    private string _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
```

```
                return _fullName;
            }
        }
    }
```

This final version evaluates the `FullName` property only when needed. If the previously calculated version is valid, it's used. If another state change invalidates the previously calculated version, it will be recalculated. Developers that use this class don't need to know the details of the implementation. None of these internal changes affect the use of the Person object. That's the key reason for using Properties to expose data members of an object.

Attaching attributes to auto-implemented properties

Field attributes can be attached to the compiler generated backing field in auto-implemented properties. For example, consider a revision to the `Person` class that adds a unique integer `Id` property. You write the `Id` property using an auto-implemented property, but your design doesn't call for persisting the `Id` property. The NonSerializedAttribute can only be attached to fields, not properties. You can attach the NonSerializedAttribute to the backing field for the `Id` property by using the `field:` specifier on the attribute, as shown in the following example:

C#

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

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

This technique works for any attribute you attach to the backing field on the auto-implemented property.

Implementing INotifyPropertyChanged

A final scenario where you need to write code in a property accessor is to support the INotifyPropertyChanged interface used to notify data binding clients that a value has changed. When the value of a property changes, the object raises the INotifyPropertyChanged.PropertyChanged event to indicate the change. The data binding libraries, in turn, update display elements based on that change. The code below shows how you would implement `INotifyPropertyChanged` for the `FirstName` property of this person class.

C#

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => _firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != _firstName)
            {
                _firstName = value;
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
        }
    }
    private string _firstName;

    public event PropertyChangedEventHandler PropertyChanged;
}
```

The `?.` operator is called the *null conditional operator*. It checks for a null reference before evaluating the right side of the operator. The end result is that if there are no subscribers to the `PropertyChanged` event, the code to raise the event doesn't execute. It would throw a `NullReferenceException` without this check in that case. For more information, see events. This example also uses the new `nameof` operator to convert from the property name symbol to its text representation. Using `nameof` can reduce errors where you've mistyped the name of the property.

Again, implementing INotifyPropertyChanged is an example of a case where you can write code in your accessors to support the scenarios you need.

Summing up

Properties are a form of smart fields in a class or object. From outside the object, they appear like fields in the object. However, properties can be implemented using the full palette of C# functionality. You can provide validation, different accessibility, lazy evaluation, or any requirements your scenarios need.