THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

ESSENTIAL C# 8.0

"Welcome to one of the most venerable and trusted franchises you could dream of in the world of C# books—and probably far beyond!"

—From the Foreword by **Mads Torgersen,** *C# Lead Designer, Microsoft*

MARK MICHAELIS
with **ERIC LIPPERT** *and*
KEVIN BOST, *Technical Editors*

IntelliTect

Essential C# 8.0

Starting in C# 6.0, there is also support for read-only, **automatically implemented properties** as follows:

Begin 6.0

```
public bool[,,] Cells { get; } = new bool[2, 3, 3];
```

This is clearly a significant improvement over the pre–C# 6.0 approach, especially given the commonality of read-only properties for something like an array of items or the `Id` in Listing 6.21.

One important note about a read-only automatically implemented property is that, like read-only fields, the compiler requires that such a property be initialized via an initializer (or in the constructor). In the preceding snippet we use an initializer, but the assignment of `Cells` from within the constructor is also permitted, as we shall see shortly.

Given the guideline that fields should not be accessed from outside their wrapping property, those programming in a C# 6.0 world will discover that there is almost never a need to use pre–C# 6.0 syntax for read-only properties; instead, the programmer can almost always use a read-only, automatically implemented property. The only exception might be when the data type of the read-only modified field does not match the data type of the property—for example, if the field was of type `int` and the read-only property was of type `double`.

Guidelines

DO create read-only properties if the property value should not be changed.

DO create read-only automatically implemented properties in C# 6.0 (or later), rather than read-only properties with a backing field if the property value should not be changed.

End 6.0

Calculated Properties

In some instances, you do not need a backing field at all. Instead, the property getter returns a calculated value, while the setter parses the value and persists it to some other member fields (if it even exists). Consider, for example, the `Name` property implementation shown in Listing 6.22. Output 6.7 shows the results.

Listing 6.22: Defining Calculated Properties

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();

      employee1.Name = "Inigo Montoya";
      System.Console.WriteLine(employee1.Name);

      // ...
  }
}
```

```
public class Employee
{
  // ...

  // FirstName property
  public string FirstName
  {
      get
      {
          return _FirstName;
      }
      set
      {
          _FirstName = value;
      }
  }
  private string _FirstName;

  // LastName property
  public string LastName
  {
      get => _LastName;
      set => _LastName = value;
  }
  private string _LastName;
  // ...

  // Name property
  public string Name
  {
      get
      {
          return $"{ FirstName } { LastName }";
      }
      set
      {
```

```
            // Split the assigned value into
            // first and last names
            string[] names;
            names = value.Split(new char[]{' '});
            if(names.Length == 2)
            {
                FirstName = names[0];
                LastName = names[1];
            }
            else
            {
                // Throw an exception if the full
                // name was not assigned
               throw new System. ArgumentException (
                    $"Assigned value '{ value }' is invalid",
                    nameof(value));[5]
            }
        }
    }
    public string Initials => $"{ FirstName[0] } { LastName[0] }";
    // ...
}
```

OUTPUT 6.7

```
Inigo Montoya
```

The getter for the `Name` property concatenates the values returned from the `FirstName` and `LastName` properties. In fact, the name value assigned is not actually stored. When the `Name` property is assigned, the value on the right side is parsed into its first and last name parts.

Access Modifiers on Getters and Setters

Begin 2.0

As previously mentioned, it is a good practice not to access fields from outside their properties because doing so circumvents any validation or additional logic that may be inserted.

An access modifier can appear on either the get or the set portion of the property implementation[6] (not on both), thereby overriding the access

5. See Advanced Block: `nameof` Operator earlier in the chapter or the full explanation in Chapter 18.

6. Introduced in C# 2.0. C# 1.0 did not allow different levels of encapsulation between the getter and setter portions of a property. It was not possible, therefore, to create a public getter and a private setter so that external classes would have read-only access to the property while code within the class could write to the property.

modifier specified on the property declaration. Listing 6.23 demonstrates how to do this.

Listing 6.23: Placing Access Modifiers on the Setter

```
class Program
{
  static void Main()
  {
      Employee employee1 = new Employee();
      employee1.Initialize(42);
      // ERROR: The property or indexer 'Employee.Id'
      // cannot be used in this context because the set
      // accessor is inaccessible
      // employee1.Id = "490";
  }
}
```

```
public class Employee
{
  public void Initialize(int id)
  {
      // Set Id property
      Id = id.ToString();
  }

  // ...
  // Id property declaration
  public string Id
  {
      get => _Id;
      // Providing an access modifier is possible in C# 2.0
      // and higher only
      private set => _Id = value;
  }
  private string _Id;

}
```

By using `private` on the setter, the property appears as read-only to classes other than `Employee`. From within `Employee`, the property appears as read/write, so you can assign the property within the class itself. When specifying an access modifier on the getter or setter, take care that the access modifier is more restrictive than the access modifier on the property

as a whole. It is a compile error, for example, to declare the property as `private` and the setter as `public`.

Guidelines

DO apply appropriate accessibility modifiers on implementations of getters and setters on all properties.

DO NOT provide set-only properties or properties with the setter having broader accessibility than the getter.

End 2.0

Properties and Method Calls Not Allowed as `ref` or `out` Parameter Values

C# allows properties to be used identically to fields, except when they are passed as `ref` or `out` parameter values. `ref` and `out` parameter values are internally implemented by passing the memory address to the target method. However, because properties can be virtual fields that have no backing field or can be read-only or write-only, it is not possible to pass the address for the underlying storage. As a result, you cannot pass properties as `ref` or `out` parameter values. The same is true for method calls. Instead, when code needs to pass a property or method call as a `ref` or `out` parameter value, the code must first copy the value into a variable and then pass the variable. Once the method call has completed, the code must assign the variable back into the property.

ADVANCED TOPIC

Property Internals

Listing 6.24 shows that getters and setters are exposed as `get_FirstName()` and `set_FirstName()` in the Common Intermediate Language (CIL).

Listing 6.24: CIL Code Resulting from Properties

```
// ...

.field private string _FirstName
.method public hidebysig specialname instance string
        get_FirstName() cil managed
```

```
{
  // Code size       12 (0xc)
  .maxstack  1
  .locals init (string V_0)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldfld      string Employee::_FirstName
  IL_0007:  stloc.0
  IL_0008:  br.s       IL_000a

  IL_000a:  ldloc.0
  IL_000b:  ret
} // End of method Employee::get_FirstName

.method public hidebysig specialname instance void
        set_FirstName(string 'value') cil managed
{
  // Code size       9 (0x9)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldarg.1
  IL_0003:  stfld      string Employee::_FirstName
  IL_0008:  ret
} // End of method Employee::set_FirstName

.property instance string FirstName()
{
  .get instance string Employee::get_FirstName()
  .set instance void Employee::set_FirstName(string)
} // End of property Employee::FirstName

// ...
```

Just as important to their appearance as regular methods is the fact that properties are an explicit construct within the CIL, too. As Listing 6.25 shows, the getters and setters are called by CIL properties, which are an explicit construct within the CIL code. Because of this, languages and compilers are not restricted to always interpreting properties based on a naming convention. Instead, CIL properties provide a means for compilers and code editors to provide special syntax.

Listing 6.25: Properties Are an Explicit Construct in CIL

```
.property instance string FirstName()
{
  .get instance string Program::get_FirstName()
  .set instance void Program::set_FirstName(string)
} // End of property Program::FirstName
```