THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

ESSENTIAL C# 7.0

"This book has been a classic for years, and remains as one of the most venerable and trusted titles in the world of C# content, and probably far beyond!"

—**Mads Torgersen**

MARK MICHAELIS

ERIC LIPPERT, *Technical Editor*

Foreword by **MADS TORGERSEN,** *C# Program Manager, Microsoft*

IntelliTect

Essential C# 7.0

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`.

End 2.0

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.

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
```

Notice in Listing 6.24 that the getters and setters that are part of the property include the `specialname` metadata. This modifier is what IDEs, such as Visual Studio, use as a flag to hide the members from IntelliSense.

Begin 3.0

An automatically implemented property is almost identical to one for which you define the backing field explicitly. In place of the manually defined backing field, the C# compiler generates a field with the name `<PropertyName>k_BackingField` in CIL. This generated field includes an attribute (see Chapter 18) called `System.Runtime.CompilerServices.CompilerGeneratedAttribute`. Both the getters and the setters are decorated with the same attribute because they, too, are generated—with the same implementation as in Listings 5.23 and 5.24.

End 3.0

Constructors

Now that you have added fields to a class and can store data, you need to consider the validity of that data. As you saw in Listing 6.3, it is possible to instantiate an object using the `new` operator. The result, however, is the ability to create an employee with invalid data. Immediately following the assignment of `employee`, you have an `Employee` object whose name and salary are not initialized. In this particular listing, you assigned the uninitialized fields immediately following the instantiation of an employee, but if you failed to do the initialization, you would not receive a warning from the compiler. As a result, you could end up with an `Employee` object with an invalid name.

Declaring a Constructor

To correct this problem, you need to provide a means of specifying the required data when the object is created. You do this using a constructor, as demonstrated in Listing 6.26.

Listing 6.26: Defining a Constructor

```
class Employee
{
  // Employee constructor
  public Employee(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }
```

```
    public string FirstName{ get; set; }
    public string LastName{ get; set; }
    public string Salary{ get; set; } = "Not Enough";

    // ...
}
```

As shown here, to define a constructor you create a method with no return type, whose method name is identical to the class name.

The constructor is the method that the runtime calls to initialize an instance of the object. In this case, the constructor takes the first name and the last name as parameters, allowing the programmer to specify these names when instantiating the `Employee` object. Listing 6.27 is an example of how to call a constructor.

Listing 6.27: Calling a Constructor

```
class Program
{
  static void Main()
  {
      Employee employee;
      employee = new Employee("Inigo", "Montoya");
      employee.Salary = "Too Little";

      System.Console.WriteLine(
          "{0} {1}: {2}",
          employee.FirstName,
          employee.LastName,
          employee.Salary);
  }
  // ...
}
```

Notice that the `new` operator returns the type of the object being instantiated (even though no return type or return statement was specified explicitly in the constructor's declaration or implementation). In addition, you have removed the initialization code for the first and last names because that initialization takes place within the constructor. In this example, you don't initialize `Salary` within the constructor, so the code assigning the salary still appears.

Developers should take care when using both assignment at declaration time and assignment within constructors. Assignments within the constructor will occur after any assignments are made when a field is

declared (such as `string Salary = "Not enough"` in Listing 6.5). Therefore, assignment within a constructor will override any value assigned at declaration time. This subtlety can lead to a misinterpretation of the code by a casual reader who assumes the value after instantiation is the one assigned in the field declaration. Therefore, it is worth considering a coding style that does not mix both declaration assignment and constructor assignment within the same class.

ADVANCED TOPIC

Implementation Details of the new Operator

Internally, the interaction between the `new` operator and the constructor is as follows. The `new` operator retrieves "empty" memory from the memory manager and then calls the specified constructor, passing a reference to the empty memory to the constructor as the implicit `this` parameter. Next, the remainder of the constructor chain executes, passing around the reference between constructors. None of the constructors have a return type; behaviorally they all return `void`. When execution of the constructor chain is complete, the `new` operator returns the memory reference, now referring to the memory in its initialized form.

Default Constructors

When you add a constructor explicitly, you can no longer instantiate an `Employee` from within `Main()` without specifying the first and last names. The code shown in Listing 6.28, therefore, will not compile.

Listing 6.28: Default Constructor No Longer Available

```
class Program
{
  static void Main()
  {
      Employee employee;
      // ERROR: No overload because method 'Employee'
      // takes '0' arguments
      employee = new Employee();

      // ...
  }
}
```

If a class has no explicitly defined constructor, the C# compiler adds one during compilation. This constructor takes no parameters and therefore is the **default constructor** by definition. As soon as you add an explicit constructor to a class, the C# compiler no longer provides a default constructor. Therefore, with `Employee(string firstName, string lastName)` defined, the default constructor, `Employee()`, is not added by the compiler. You could manually add such a constructor, but then you would again be allowing construction of an `Employee` without specifying the employee name.

It is not necessary to rely on the default constructor defined by the compiler. It is also possible for programmers to define a default constructor explicitly—perhaps one that initializes some fields to particular values. Defining the default constructor simply involves declaring a constructor that takes no parameters.

Object Initializers

Begin 3.0

Starting with C# 3.0, the C# language team added functionality to initialize an object's accessible fields and properties using an **object initializer**. The object initializer consists of a set of member initializers enclosed in curly braces following the constructor call to create the object. Each member initializer is the assignment of an accessible field or property name with a value (see Listing 6.29).

Listing 6.29: Calling an Object Initializer

```
class Program
{
  static void Main()
  {
        Employee employee1 = new Employee("Inigo", "Montoya")
            { Title = "Computer Nerd", Salary = "Not enough"};
      // ...
  }
}
```

Notice that the same constructor rules apply even when using an object initializer. In fact, the resultant CIL is exactly the same as it would be if the fields or properties were assigned within separate statements immediately following the constructor call. The order of member initializers in C# provides the sequence for property and field assignment in the statements following the constructor call within CIL.