THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

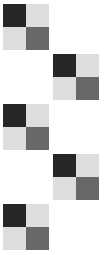
ESSENTIAL C# 12.0

"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,** *Principal Architect, Microsoft*

MARK MICHAELIS
with
KEVIN BOST, *Technical Editor*

IntelliTect



Essential C# 12.0

Mark Michaelis
with Kevin Bost,
Technical Editor

Addison-Wesley
Hoboken, New Jersey

LISTING 6.27: Calling a Constructor

```
public class Program
{
    public static void Main()
    {
        Employee employee;
        employee = new("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. In addition, the initialization for the first and last names occurs via the property initializers that execute during construction. In this example, you don't initialize `Salary` within the constructor, so the code assigning the salary still appears.

End 12.0

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.

Defining a Constructor

It is also possible to define constructors separately from the type declaration. As shown in Listing 6.28, to define a (non-primary) constructor, you create a method with no return type, whose method name is identical to the type name. Even though no return type or return statement was specified in the constructor's declaration or implementation, invoking the constructor (via the new operator), still returns an instance of the type.

Listing 6.28: Defining a Constructor

```
public 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";

    // ...
}
```

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 property or field is declared (such as `string Salary { get; set; } = "Not enough"` in Listing 6.26). 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 property or field declaration. Therefore, it is worth considering a coding style that does not mix both declaration assignment and constructor assignment for the same field or property

Default and Copy 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 6.29, therefore, will not compile.

Listing 6.29: Default Constructor No Longer Available

```
public class Program
{
    public 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 so, by definition, it is the **default constructor**. 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's name.

It is not necessary to rely on the default constructor defined by the compiler. That is, programmers can 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.

A **copy constructor** is a constructor that takes a single parameter of the containing type. For example:

```
public Employee(Employee original)
{
// Copy properties between employees here.
}
```

Constructors like this are useful for cloning an instance of an object into a new duplicate instance.

Object Initializers

To initialize an object's accessible fields and properties, you can use the concept of an **object initializer**—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 6.30).

LISTING 6.30: Calling an Object Initializer with Explicit Member Assignment

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

Notice that the same constructor rules apply even when using an object initializer. The order of member initializers in C# provides the sequence for property and field assignment in the statements following the constructor call within CIL.

In general, all properties should be initialized to reasonable default values by the time the constructor exits. Moreover, by using validation logic on the setter, it is possible to restrict the assignment of invalid data to a property. On occasion, the values on one or more properties may cause other properties on the same object to contain invalid values. When this occurs, exceptions from the invalid state should be postponed until the invalid interrelated property values become relevant.

Guidelines

DO provide sensible defaults for all properties, ensuring that defaults do not result in a security hole or significantly inefficient code.

DO allow properties to be set in any order, even if this results in a temporarily invalid object state.

ADVANCED TOPIC

Collection Initializers

Using a similar syntax to that of object initializers, **collection initializers**[8] support a similar feature set as object initializers, only with collections. Specifically, a collection initializer allows the assignment of items within the collection at the time of the collection's instantiation. Borrowing the same syntax used for arrays, the collection initializer initializes each item within the collection as part of collection creation. Initializing a list of `Employee` objects, for example, involves specifying each item within curly braces following the constructor call, as shown in Listing 6.31.

Listing 6.31: Calling an Object Initializer

```
public class Program
{
    public static void Main()
    {
        List<Employee> employees = new()
            {
                new("Inigo", "Montoya"),
                new("Kevin", "Bost")
            };
        // ...
    }
}
```

After the assignment of a new collection instance, the compiler-generated code instantiates each object in sequence and adds them to the collection via the `Add()` method.

8. Added in C# 3.0.

Begin 9.0

Init Only Setters

Object initializers allow for specifying member values during object initialization. Any read-only properties cannot be set this way, however, because properties with only getters may only be set during object construction and object initializers runs after this. To address the problem, C# 9.0 added support for init only setters, which can be set from within object initializers but not afterward. Listing 6.32 demonstrates the use of `init` only setter for the `Salary` property.

LISTING 6.32: Init Only Setter

```
public class Employee
{
    public Employee(int id, string name)
    {
        Id = id;
        Name = name;
        Salary = null;
    }

    // ...

    public int Id { get; }
    public string Name { get; }

    public string? Salary
    {
        get => _Salary;
        init => _Salary = value;
    }
    private string? _Salary;
}

public class Program
{
    public static void Main()
    {
        Employee employee = new(42, "Inigo Montoya")
        {
            Salary = "Sufficient"
        };

        // ERROR:  Property or indexer 'Employee.Salary'
        // cannot be assigned after initialization completes.
```