CHAPTER 4

Essential Language Features

C# is a feature-rich language and not all programmers are familiar with all of the features I rely on in this book. In this chapter, I describe the C# language features that a good MVC programmer needs to know and that I use in examples throughout this book.

I provide only a short summary of each feature. If you want more in-depth coverage of C# or LINQ, three of my books may be of interest. For a complete guide to C#, try *Introducing Visual C#;* for in-depth coverage of LINQ, check out *Pro LINQ in C#*; and for a detailed examination of the .NET support for asynchronous programming see *Pro .Net Parallel Programming in C#*. All of these books are published by Apress. Table 4-1 provides the summary for this chapter.

Table 4-1. *Chapter Summary*

Problem	Solution	Listing
Simplify C# properties	Use automatically implemented properties	1–7
Create an object and sets its properties in a single step	Use an object or collection initializer	8–10
Add functionality to a class which cannot be modified	Use an extension method	11–18
Simplify the use of delegates	Use a lambda expression	19–23
Use implicit typing	Use the `var` keyword	24
Create objects without defining a type	Use an anonymous type	25–26
Query collections of objects as though there were a database	Use LINQ	27–31
Simplify the use of asynchronous methods	Use the `async` and `await` keywords	32–33

Preparing the Example Project

To demonstrate the language features in this part of the book, I have created a new Visual Studio project called `LanguageFeatures` using the `ASP.NET MVC Web Application` template. I selected the `Empty` option for the initial content and checked the option for `MVC` folders and references, just as I did in Chapter 2. The language features that I describe in this chapter are not specific to MVC, but Visual Studio Express 2013 for Web doesn't support creating projects that can write to the console, so you will have to create an MVC app if you want to follow along with the examples. I need a simple controller to demonstrate these language features, so I created the `HomeController.cs` file in the `Controllers` folder–I did this by right-clicking on the `Controllers` folder in the Solution Explorer, selecting `Add` ➤ `Controller` from the pop-up menu, selecting `MVC 5 Controller-Empty` from the `Add Scaffold` menu, and clicking the `Add` button. I set the name to `HomeController` in the `Add Controller` dialog and clicked the `Add` button to create the controller class file, the edited contents of which you can see in Listing 4-1.

Listing 4-1. The Initial Content of the HomeController.cs File

```
using System;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }
    }
}
```

I will create action methods for each example, so the result from the Index action method is a basic message to keep the project simple.

■ **Caution** The HomeController class won't compile at the moment because it imports the LanguageFeatures.Models namespace. This namespace won't create until I add a class to the Models folder, which I do as part of the first example in the next section.

To display the results from my action methods, I right-clicked the Index action method, selected Add View and created a new view called Result. You can see the contents of the view file in Listing 4-2. (It doesn't matter which options you select in the Add View dialog because you will replace the initial content of the file with the markup shown in the listing).

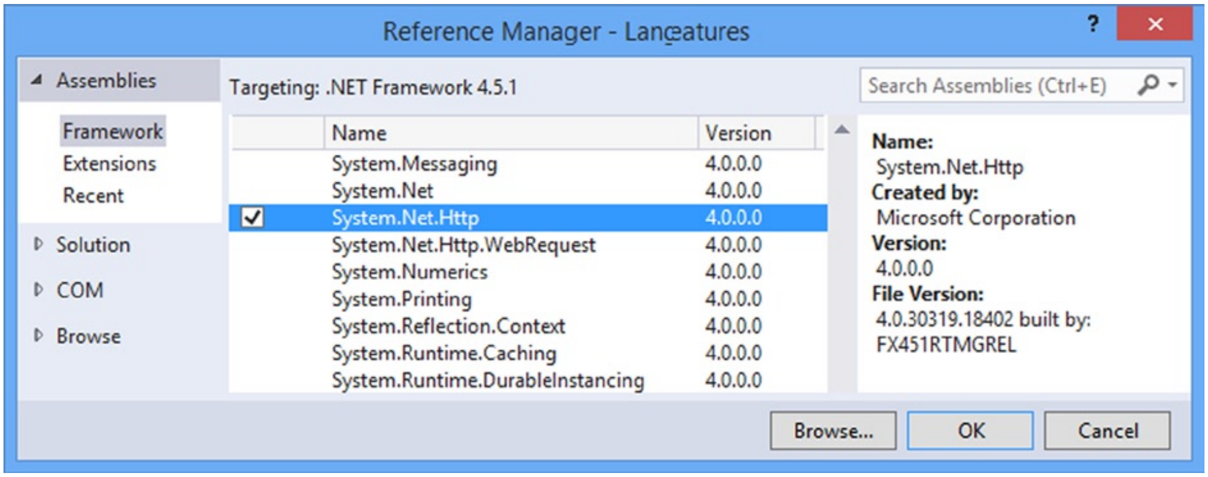
Listing 4-2. The Contents of the Result.cshtml File

```
@model String

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Result</title>
</head>
<body>
    <div>
        @Model
    </div>
</body>
</html>
```

You can see that this is a strongly typed view, where the model type is String–for the most part, the examples that follow are not complex examples and I can represent the results as a simple string.

Adding the System.Net.Http Assembly

Later in the chapter, I'll be using an example that relies on the `System.Net.Http` assembly, which isn't added to MVC projects by default. Select `Add Reference` from the Visual Studio `Project` menu to open the `Reference Manager` window. Ensure that the `Assemblies` section is selected on the left-hand side and locate and check the `System.Net.Http` item, as shown in Figure 4-1.



Targeting: .NET Framework 4.5.1

	Name	Version
	System.Messaging	4.0.0.0
	System.Net	4.0.0.0
✓	System.Net.Http	4.0.0.0
	System.Net.Http.WebRequest	4.0.0.0
	System.Numerics	4.0.0.0
	System.Printing	4.0.0.0
	System.Reflection.Context	4.0.0.0
	System.Runtime.Caching	4.0.0.0
	System.Runtime.DurableInstancing	4.0.0.0

Name:
System.Net.Http
Created by:
Microsoft Corporation
Version:
4.0.0.0
File Version:
4.0.30319.18402 built by:
FX451RTMGREL

Figure 4-1. *Adding an assembly to the project*

Using Automatically Implemented Properties

The regular C# property feature lets you expose a piece of data from a class in a way that decouples the data from how it is set and retrieved. Listing 4-3 contains a simple example in a class called `Product`, which I added to the `Models` folder of the `LanguageFeatures` project in a class file called `Product.cs`

Listing 4-3. Defining a Property in the Product.cs File

```
namespace LanguageFeatures.Models {
    public class Product {
        private string name;

        public string Name {
            get { return name; }
            set { name = value; }
        }
    }
}
```

The property, called `Name`, is shown in bold. The statements in the `get` code block (known as the *getter*) are performed when the value of the property is read, and the statements in the `set` code block (known as the *setter*) are performed when a value is assigned to the property (the special variable `value` represents the assigned value). A property is consumed by other classes as though it were a field, as shown in Listing 4-4, which shows an `AutoProperty` action method I added to the `Home` controller.

Listing 4-4. Consuming a Property in the HomeController.cs File

```
using System;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        public ViewResult AutoProperty() {
            // create a new Product object
            Product myProduct = new Product();

            // set the property value
            myProduct.Name = "Kayak";

            // get the property
            string productName = myProduct.Name;

            // generate the view
            return View("Result",
                (object)String.Format("Product name: {0}", productName));
        }
    }
}
```

You can see that the property value is read and set just like a regular field. Using properties is preferable to using fields because you can change the statements in the `get` and `set` blocks without needing to change the classes that depend on the property.

■ **Tip** You may notice that I cast the second argument to the `View` method to an `object` in Listing 4-4. This is because the `View` method has an overload that accepts two `String` arguments and which has a different meaning to the overload that accepts a `String` and an `object`. To avoid calling the wrong one, I explicitly cast the second argument. I return to the `View` method and its overloads in Chapter 20.

You can see the effect of this example by starting the project and navigating to `/Home/AutoProperty` (which targets the `AutoProperty` action method and will be the pattern for testing each example in this chapter). Because I pass a string from the action method to the view, I am going to show you the results as text, rather than a screen shot. Here is the result of targeting the action method in Listing 4-4:

```
Product name: Kayak
```

Properties are all well and good, but they become tedious when you have a class that has a lot of properties, all of which mediate access to a field, producing a class file that is needlessly verbose, as shown in Listing 4-5, which shows some additional properties I added to the `Product` class in the `Product.cs` file.

Listing 4-5. Verbose Property Definitions in the Product.cs File

```
namespace LanguageFeatures.Models {

    public class Product {
        private int productID;
        private string name;
        private string description;
        private decimal price;
        private string category;

        public int ProductID {
            get { return productID; }
            set { productID = value; }
        }

        public string Name {
            get { return name; }
            set { name = value; }
        }

        public string Description {
            get { return description; }
            set { description = value; }
        }

        //...and so on...
    }
}
```

What I want is the flexibility of properties without having to duplicate the getters and setters. The solution is an *automatically implemented property*, also known as an *automatic property*. With an automatic property, you can create the pattern of a field-backed property, without defining the field or specifying the code in the getter and setter, as Listing 4-6 shows.

Listing 4-6. Using Automatically Implemented Properties in the Product.cs File

```
namespace LanguageFeatures.Models {

    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

Notice that I do not define the bodies of the getter and setter or the field that the property is backed by. Both of these are done for me by the C# compiler when the class is compiled. Using an automatic property is no different from using a regular property; the code in the action method in Listing 4-4 will work without any modification.

By using automatic properties, I save myself some typing, create code that is easier to read, but still preserve the flexibility that a property provides. If the day comes when I need to change the way a property is implemented, I can return to the regular property format. As a demonstration, Listing 4-7 shows what I would have to do if I needed to change the way the Name property is composed.

Listing 4-7. Reverting from an Automatic to a Regular Property in the Product.cs File

```
namespace LanguageFeatures.Models {

    public class Product {
        private string name;

        public int ProductID { get; set; }

        public string Name {
            get {
                return ProductID + name;
            }
            set {
                name = value;
            }
        }

        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { set; get; }
    }
}
```

■ **Note** Notice that I must implement both the getter and setter to return to a regular property. C# does not support mixing automatic- and regular-style getters and setters in a single property.

Using Object and Collection Initializers

Another tiresome programming task is constructing a new object and then assigning values to the properties, as illustrated by Listing 4-8, which shows the addition of a CreateProduct action method to the Home controller.

Listing 4-8. Constructing and Initializing an Object with Properties in the HomeController.cs File

```
using System;
using System.Web.Mvc;
using LanguageFeatures.Models;
```

```
namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        public ViewResult AutoProperty() {
            // ...statements omitted for brevity...
        }

        public ViewResult CreateProduct() {

            // create a new Product object
            Product myProduct = new Product();

            // set the property values
            myProduct.ProductID = 100;
            myProduct.Name = "Kayak";
            myProduct.Description = "A boat for one person";
            myProduct.Price = 275M;
            myProduct.Category = "Watersports";

            return View("Result",
                (object)String.Format("Category: {0}", myProduct.Category));
        }
    }
}
```

I go through three stages to create a Product object and produce a result: create the object, set the parameter values, and then call the View method so I can display the result through the view. Fortunately, I can use the *object initializer* feature, which allows me to create and populate the Product instance in a single step, as shown in Listing 4-9.

Listing 4-9. Using the Object Initializer Feature in the HomeController.cs File

```
...
public ViewResult CreateProduct() {

    // create and populate a new Product object
    Product myProduct = new Product {
        ProductID = 100, Name = "Kayak",
        Description = "A boat for one person",
        Price = 275M, Category = "Watersports"
    };

    return View("Result",
        (object)String.Format("Category: {0}", myProduct.Category));
}
...
```

The braces ({}) after the call to the Product name form the *initializer*, which I use to supply values to the parameters as part of the construction process. The same feature let me initialize the contents of collections and arrays as part of the construction process, as demonstrated by Listing 4-10.

Listing 4-10. Initializing Collections and Arrays in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult CreateCollection() {
            string[] stringArray = { "apple", "orange", "plum" };

            List<int> intList = new List<int> { 10, 20, 30, 40 };

            Dictionary<string, int> myDict = new Dictionary<string, int> {
                { "apple", 10 }, { "orange", 20 }, { "plum", 30 }
            };

            return View("Result", (object)stringArray[1]);
        }
    }
}
```

The listing demonstrates how to construct and initialize an array and two classes from the generic collection library. This feature is a syntax convenience—it just makes C# more pleasant to use but does not have any other impact or benefit.

Using Extension Methods

Extension methods are a convenient way of adding methods to classes that you do not own and cannot modify directly. Listing 4-11 shows a `ShoppingCart` class, which I added to the `Models` folder in a file called `ShoppingCart.cs` file and which represents a collection of `Product` objects.

Listing 4-11. The ShoppingCart Class in the ShoppingCart.cs File

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart {
        public List<Product> Products { get; set; }
    }
}
```

This is a simple class that acts as a wrapper around a List of Product objects (I only need a basic class for this example). Suppose I need to be able to determine the total value of the Product objects in the ShoppingCart class, but I cannot modify the class itself, perhaps because it comes from a third party and I do not have the source code. I can use an extension method to add the functionality I need. Listing 4-12 shows the MyExtensionMethods class that I added to the Models folder in the MyExtensionMethods.cs file.

Listing 4-12. Defining an Extension Method in the MyExtensionMethods.cs File

```
namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this ShoppingCart cartParam) {
            decimal total = 0;
            foreach (Product prod in cartParam.Products) {
                total += prod.Price;
            }
            return total;
        }
    }
}
```

The this keyword in front of the first parameter marks TotalPrices as an extension method. The first parameter tells .NET which class the extension method can be applied to—ShoppingCart in this case. I can refer to the instance of the ShoppingCart that the extension method has been applied to by using the cartParam parameter. My method enumerates the Products in the ShoppingCart and returns the sum of the Product.Price property. Listing 4-13 shows how I apply an extension method in a new action method called UseExtension I added to the Home controller.

Listing 4-13. Applying an Extension Method in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult UseExtension() {
            // create and populate ShoppingCart
            ShoppingCart cart = new ShoppingCart {
                Products = new List<Product> {
                    new Product {Name = "Kayak", Price = 275M},
                    new Product {Name = "Lifejacket", Price = 48.95M},
                    new Product {Name = "Soccer ball", Price = 19.50M},
                    new Product {Name = "Corner flag", Price = 34.95M}
                }
            };
            // get the total value of the products in the cart
            decimal cartTotal = cart.TotalPrices();

            return View("Result",
                (object)String.Format("Total: {0:c}", cartTotal));
        }
    }
}
```

■ **Note** Extension methods do not let you break through the access rules that classes define for their methods, fields, and properties. You can extend the functionality of a class by using an extension method, but only using the class members that you had access to anyway.

The key statement is this one:

```
...
decimal cartTotal = cart.TotalPrices();
...
```

I call the `TotalPrices` method on a `ShoppingCart` object as though it were part of the `ShoppingCart` class, even though it is an extension method defined by a different class altogether. .NET will find extension classes if they are in the scope of the current class, meaning that they are part of the same namespace or in a namespace that is the subject of a `using` statement. Here is the result from the `UseExtension` action method, which you can see by starting the application and navigating to the `/Home/UseExtension` URL:

```
Total: $378.40
```

Applying Extension Methods to an Interface

I can also create extension methods that apply to an interface, which allows me to call the extension method on all of the classes that implement the interface. Listing 4-14 shows the `ShoppingCart` class updated to implement the `IEnumerable<Product>` interface.

Listing 4-14. Implementing an Interface in the ShoppingCart.cs File

```
using System.Collections;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart: IEnumerable<Product> {

        public List<Product> Products { get; set; }
        public IEnumerator<Product> GetEnumerator() {
            return Products.GetEnumerator();
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
    }
}
```

I can now update my extension method so that it deals with `IEnumerable<Product>`, as shown in Listing 4-15.

Listing 4-15. An Extension Method That Works on an Interface in the MyExtensionMethods.cs File

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumerable<Product> productEnum) {
            decimal total = 0;
            foreach (Product prod in productEnum) {
                total += prod.Price;
            }
            return total;
        }
    }
}
```

The first parameter type has changed to `IEnumerable<Product>`, which means that the `foreach` loop in the method body works directly on `Product` objects. The switch to the interface means that I can calculate the total value of the `Product` objects enumerated by any `IEnumerable<Product>`, which includes instances of `ShoppingCart` but also arrays of `Products`, as shown in Listing 4-16.

Listing 4-16. Extension Methods Applies to Implementations of an Interface in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }
        // ...other action methods omitted for brevity...

        public ViewResult UseExtensionEnumerable() {

            IEnumerable<Product> products = new ShoppingCart {
                Products = new List<Product> {
                    new Product {Name = "Kayak", Price = 275M},
                    new Product {Name = "Lifejacket", Price = 48.95M},
                    new Product {Name = "Soccer ball", Price = 19.50M},
                    new Product {Name = "Corner flag", Price = 34.95M}
                }
            };

            // create and populate an array of Product objects
            Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M},
                new Product {Name = "Corner flag", Price = 34.95M}
            };

            // get the total value of the products in the cart
            decimal cartTotal = products.TotalPrices();
            decimal arrayTotal = products.TotalPrices();

            return View("Result",
                (object)String.Format("Cart Total: {0}, Array Total: {1}",
                    cartTotal, arrayTotal));
        }
    }
}
```

■ **Note** The way that C# arrays implement the `IEnumerable<T>` interface is a little unusual. You will not find it included in the list of implemented interfaces in the MSDN documentation. The support is handled by the compiler so that code for earlier versions C# will still compile. Odd, but true. I could have used another generic collection class in this example, but I wanted to show off my knowledge of the dark corners of the C# specification. Also odd, but true.

If you start the project and target the action method, you will see the following results, which demonstrate that I get the same result from the extension method, irrespective of how the `Product` objects are collected:

```
Cart Total: 378.40, Array Total: 378.40
```

Creating Filtering Extension Methods

The last thing I want to show you about extension methods is that they can be used to filter collections of objects. An extension method that operates on an `IEnumerable<T>` and that also returns an `IEnumerable<T>` can use the `yield` keyword to apply selection criteria to items in the source data to produce a reduced set of results. Listing 4-17 demonstrates such a method, which I have added to the `MyExtensionMethods` class.

Listing 4-17. A Filtering Extension Method in the MyExtensionMethods.cs File

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumerable<Product> productEnum) {
            decimal total = 0;
            foreach (Product prod in productEnum) {
                total += prod.Price;
            }
            return total;
        }

        public static IEnumerable<Product> FilterByCategory(
                this IEnumerable<Product> productEnum, string categoryParam) {

            foreach (Product prod in productEnum) {
                if (prod.Category == categoryParam) {
                    yield return prod;
                }
            }
        }
    }
}
```

This extension method, called `FilterByCategory`, takes an additional parameter that allows me to inject a filter condition when I call the method. Those `Product` objects whose `Category` property matches the parameter are returned in the result `IEnumerable<Product>` and those that do not match are discarded. Listing 4-18 shows this method being used.

Listing 4-18. Using the Filtering Extension Method in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {

    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }
        // ... other action methods omitted for brevity...

        public ViewResult UseFilterExtensionMethod() {

            IEnumerable<Product> products = new ShoppingCart {
                Products = new List<Product> {
                    new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
                    new Product {Name = "Lifejacket", Category = "Watersports",
                       Price = 48.95M},
                    new Product {Name = "Soccer ball", Category = "Soccer",
                       Price = 19.50M},
                    new Product {Name = "Corner flag", Category = "Soccer",
                       Price = 34.95M}
                }
            };

            decimal total = 0;
            foreach (Product prod in products.FilterByCategory("Soccer")) {
                total += prod.Price;
            }

            return View("Result", (object)String.Format("Total: {0}", total));
        }
    }
}
```

When I call the FilterByCategory method on the ShoppingCart, only those Products in the Soccer category are returned. If you start the project and target the UseFilterExtensionMethod action method, you will see the following result, which is the sum of the Soccer product prices:

```
Total: 54.45
```

Using Lambda Expressions

I can use a delegate to make my FilterByCategory method more general. That way, the delegate that will be invoked against each Product can filter the objects in any way I choose, as illustrated by Listing 4-19, which shows the Filter extension method I added to the MyExtensionMethods class.

Listing 4-19. Using a Delegate in an Extension Method in the MyExtensionMethods.cs File

```
using System;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {
        public static decimal TotalPrices(this IEnumerable<Product> productEnum) {
            decimal total = 0;
            foreach (Product prod in productEnum) {
                total += prod.Price;
            }
            return total;
        }

        public static IEnumerable<Product> FilterByCategory(
                this IEnumerable<Product> productEnum, string categoryParam) {

            foreach (Product prod in productEnum) {
                if (prod.Category == categoryParam) {
                    yield return prod;
                }
            }
        }

        public static IEnumerable<Product> Filter(
              this IEnumerable<Product> productEnum, Func<Product, bool> selectorParam) {

            foreach (Product prod in productEnum) {
                if (selectorParam(prod)) {
                    yield return prod;
                }
            }
        }
    }
}
```

I used a Func as the filtering parameter, which means that I do not need to define the delegate as a type. The delegate takes a Product parameter and returns a bool, which will be true if that Product should be included in the results. The other end of this arrangement is a little verbose, as illustrated by Listing 4-20, which shows the changes I made to the UseFilterExtensionMethod action method in the Home controller.

Listing 4-20. Using the Filtering Extension Method with a Func in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    // create and populate ShoppingCart
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };
    Func<Product, bool> categoryFilter = delegate(Product prod) {
        return prod.Category == "Soccer";
    };

    decimal total = 0;

    foreach (Product prod in products.Filter(categoryFilter)) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...
```

I have taken a step forward, in the sense that I can now filter the `Product` objects using any criteria specified in the delegate, but I must define a `Func` for each kind of filtering that I want, which is not ideal. The less verbose alternative is to use a *lambda expression*, which is a concise format for expressing a method body in a delegate. I can use it to replace my delegate definition in the action method, as shown in Listing 4-21.

Listing 4-21. Using a Lambda Expression to Replace a Delegate Definition in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    // create and populate ShoppingCart
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };

    Func<Product, bool> categoryFilter = prod => prod.Category == "Soccer";

    decimal total = 0;

    foreach (Product prod in products.Filter(categoryFilter)) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...
```

The lambda expression is shown in bold. The parameter is expressed without specifying a type, which will be inferred automatically. The `=>` characters are read aloud as "goes to" and links the parameter to the result of the lambda expression. In my example, a `Product` parameter called `prod` goes to a `bool` result, which will be `true` if the `Category` parameter of `prod` is equal to `Soccer`. I can make my syntax even tighter by doing away with the `Func` entirely, as shown in Listing 4-22.

Listing 4-22. A Lambda Expression Without a Func in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };

    decimal total = 0;

    foreach (Product prod in products.Filter(prod => prod.Category == "Soccer")) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...
```

In this example, I supplied the lambda expression as the parameter to the `Filter` method. This is a nice and natural way of expressing the filter I want to apply. I can combine multiple filters by extending the result part of the lambda expression, as shown in Listing 4-23.

Listing 4-23. Extending the Filtering Expressed by the Lambda Expression in the HomeController.cs File

```
...
public ViewResult UseFilterExtensionMethod() {

    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };

    decimal total = 0;

    foreach (Product prod in products
            .Filter(prod => prod.Category == "Soccer" || prod.Price > 20)) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...
```

This revised lambda expression will match `Product` objects that are in the `Soccer` category or whose `Price` property is greater than 20.

OTHER FORMS FOR LAMBDA EXPRESSIONS

I don't need to express the logic of my delegate in the lambda expression. I can as easily call a method, like this:

```
prod => EvaluateProduct(prod)
```

If I need a lambda expression for a delegate that has multiple parameters, I must wrap the parameters in parentheses, like this:

```
(prod, count) => prod.Price > 20 && count > 0
```

And, finally, if I need logic in the lambda expression that requires more than one statement, I can do so by using braces ({}) and finishing with a `return` statement, like this:

```
(prod, count) => {
    //...multiple code statements
    return result;
}
```

You do not need to use lambda expressions in your code, but they are a neat way of expressing complex functions simply and in a manner that is readable and clear. I like them a lot, and you will see them used liberally throughout this book.

Using Automatic Type Inference

The C# `var` keyword allows you to define a local variable without explicitly specifying the variable type, as demonstrated by Listing 4-24. This is called *type inference,* or *implicit typing*.

Listing 4-24. Using Type Inference

```
..
var myVariable = new Product { Name = "Kayak", Category = "Watersports", Price = 275M };

string name = myVariable.Name;  // legal
int count = myVariable.Count;   // compiler error
...
```

It is not that `myVariable` does not have a type. It is just that I am asking the compiler to infer it from the code. You can see from the statements that follow that the compiler will allow only members of the inferred class—`Product` in this case—to be called.

Using Anonymous Types

By combining object initializers and type inference, I can create simple data-storage objects without needing to define the corresponding class or struct. Listing 4-25 shows an example.

Listing 4-25. Creating an Anonymous Type

```
...
var myAnonType = new {
    Name = "MVC",
    Category = "Pattern"
};

...
```

In this example, `myAnonType` is an anonymously typed object. This does not mean that it is dynamic in the sense that JavaScript variables are dynamic. It just means that the type definition will be created automatically by the compiler. Strong typing is still enforced. You can get and set only the properties that have been defined in the initializer, for example.

The C# compiler generates the class based on the name and type of the parameters in the initializer. Two anonymously typed objects that have the same property names and types will be assigned to the same automatically generated class. This means I can create arrays of anonymously typed objects, as demonstrated by Listing 4-26, which shows the `CreateAnonArray` action method I added to the `Home` controller.

Listing 4-26. Creating an Array of Anonymously Typed Objects in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult CreateAnonArray() {

            var oddsAndEnds = new[] {
                new { Name = "MVC", Category = "Pattern"},
                new { Name = "Hat", Category = "Clothing"},
                new { Name = "Apple", Category = "Fruit"}
            };

            StringBuilder result = new StringBuilder();
            foreach (var item in oddsAndEnds) {
                result.Append(item.Name).Append(" ");
            }

            return View("Result", (object)result.ToString());
        }
    }
}
```

Notice that I use `var` to declare the variable array. I must do this because I do not have a type to specify, as I would in a regularly typed array. Even though I have not defined a class for any of these objects, I can still enumerate the contents of the array and read the value of the `Name` property from each of them. This is important, because without this feature, I would not be able to create arrays of anonymously typed objects at all. Or, rather, I *could* create the arrays, but I would not be able to do anything useful with them. You will see the following results if you run the example and target the action method:

```
MVC Hat Apple
```

Performing Language Integrated Queries

All of the features I have described so far are put to good use in LINQ. I love LINQ. It is a wonderful and compelling addition to .NET. If you have never used LINQ, you have been missing out. LINQ is a SQL-like syntax for querying data in classes. Imagine that I have a collection of `Product` objects, and I want to find the three highest prices and pass them to the `View` method. Without LINQ, I would end up with something similar to Listing 4-27, which shows the `FindProducts` action method I added to the `Home` controller.

Listing 4-27. Querying Without LINQ in the HomeController.cs File

```
...
public ViewResult FindProducts() {

    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };

    // define the array to hold the results
    Product[] foundProducts = new Product[3];
    // sort the contents of the array
    Array.Sort(products, (item1, item2) => {
        return Comparer<decimal>.Default.Compare(item1.Price, item2.Price);
    });
    // get the first three items in the array as the results
    Array.Copy(products, foundProducts, 3);

    // create the result
    StringBuilder result = new StringBuilder();
    foreach (Product p in foundProducts) {
        result.AppendFormat("Price: {0} ", p.Price);
    }

    return View("Result", (object)result.ToString());
}
...
```

With LINQ, I can significantly simplify the querying process, as demonstrated in Listing 4-28.

Listing 4-28. Using LINQ to Query Data in the HomeController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web.Mvc;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Navigate to a URL to show an example";
        }

        // ...other action methods omitted for brevity...

        public ViewResult FindProducts() {

            Product[] products = {
             new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
             new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
             new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
             new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
            };

            var foundProducts = from match in products
                                orderby match.Price descending
                                select new { match.Name, match.Price };

            // create the result
            int count = 0;
            StringBuilder result = new StringBuilder();
            foreach (var p in foundProducts) {
                result.AppendFormat("Price: {0} ", p.Price);
                if (++count == 3) {
                    break;
                }
            }

            return View("Result", (object)result.ToString());
        }
    }
}
```

This is a lot neater. You can see the SQL-like query shown in bold. I order the `Product` objects in descending order and use the `select` keyword to return an anonymous type that contains just the `Name` and `Price` properties. This style of LINQ is known as *query syntax*, and it is the kind that developers find most comfortable when they start using LINQ. The wrinkle in this query is that it returns one anonymously typed object for every `Product` in the array that I used in the source query, so I need to play around with the results to get the first three and print out the details.

However, if you are willing to forgo the simplicity of the query syntax, you can get a lot more power from LINQ. The alternative is the *dot-notation syntax*, or *dot notation*, which is based on extension methods. Listing 4-29 shows this alternative syntax used to process the Product objects.

Listing 4-29. Using LINQ Dot Notation in the HomeController.cs File

```
...
public ViewResult FindProducts() {

    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };

    var foundProducts = products.OrderByDescending(e => e.Price)
                            .Take(3)
                            .Select(e => new { e.Name, e.Price });

    StringBuilder result = new StringBuilder();
    foreach (var p in foundProducts) {
        result.AppendFormat("Price: {0} ", p.Price);
    }

    return View("Result", (object)result.ToString());
}
...
```

This LINQ query, shown in bold, is not as nice to look at as the one expressed in query syntax, but not all LINQ features have corresponding C# keywords. For serious LINQ queries, I need to switch to using extension methods. Each of the LINQ extension methods in the listing is applied to an IEnumerable<T> and returns an IEnumerable<T> too, which allows me to chain the methods together to form complex queries.

■ **Note** All of the LINQ extension methods are in the System.Linq namespace, which you must bring into scope with a using statement before you can make queries. Visual Studio adds the System.Linq namespace to controller classes automatically, but you may need to add it manually elsewhere in an MVC project.

The OrderByDescending method rearranges the items in the data source. In this case, the lambda expression returns the value I want used for comparisons. The Take method returns a specified number of items from the front of the results (this is what I couldn't do using query syntax). The Select method allows me to project my results, specifying the structure I want. In this case, I am projecting an anonymous object that contains the Name and Price properties.

■ **Tip** Notice that I have not needed to specify the names of the properties in the anonymous type. C# has inferred this from the properties I picked in the Select method.

Table 4-2 describes the most useful LINQ extension methods. I use LINQ liberally throughout the rest of this book, and you may find it useful to return to this table when you see an extension method that you have not encountered before. All of the LINQ methods shown in the table operate on `IEnumerable<T>`.

Table 4-2. *Some Useful LINQ Extension Methods*

Extension Method	Description	Deferred
`All`	Returns `true` if all the items in the source data match the predicate	No
`Any`	Returns `true` if at least one of the items in the source data matches the predicate	No
`Contains`	Returns `true` if the data source contains a specific item or value	No
`Count`	Returns the number of items in the data source	No
`First`	Returns the first item from the data source	No
`FirstOrDefault`	Returns the first item from the data source or the default value if there are no items	No
`Last`	Returns the last item in the data source	No
`LastOrDefault`	Returns the last item in the data source or the default value if there are no items	No
`Max` `Min`	Returns the largest or smallest value specified by a lambda expression	No
`OrderBy` `OrderByDescending`	Sorts the source data based on the value returned by the lambda expression	Yes
`Reverse`	Reverses the order of the items in the data source	Yes
`Select`	Projects a result from a query	Yes
`SelectMany`	Projects each data item into a sequence of items and then concatenates all of those resulting sequences into a single sequence	Yes
`Single`	Returns the first item from the data source or throws an exception if there are multiple matches	No
`SingleOrDefault`	Returns the first item from the data source or the default value if there are no items, or throws an exception if there are multiple matches	No
`Skip` `SkipWhile`	Skips over a specified number of elements, or skips while the predicate matches	Yes
`Sum`	Totals the values selected by the predicate	No
`Take` `TakeWhile`	Selects a specified number of elements from the start of the data source or selects items while the predicate matches	Yes
`ToArray` `ToDictionary` `ToList`	Converts the data source to an array or other collection type	No
`Where`	Filters items from the data source that do not match the predicate	Yes

Understanding Deferred LINQ Queries

You will notice that Table 2 includes a `Deferred` column. There is an interesting variation in the way that the extension methods are executed in a LINQ query. A query that contains only deferred methods is not executed until the items in the result are enumerated, as demonstrated by Listing 4-30, which shows a simple change to the `FindProducts` action method.

Listing 4-30. Using Deferred LINQ Extension Methods in a Query in the HomeController.cs File

```
...
public ViewResult FindProducts() {

    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };

    var foundProducts = products.OrderByDescending(e => e.Price)
                            .Take(3)
                            .Select(e => new {
                                e.Name,
                                e.Price
                            });

    products[2] = new Product { Name = "Stadium", Price = 79600M };

    StringBuilder result = new StringBuilder();
    foreach (var p in foundProducts) {
        result.AppendFormat("Price: {0} ", p.Price);
    }

    return View("Result", (object)result.ToString());
}
...
```

Between defining the LINQ query and enumerating the results, I changed one of the items in the `products` array. The output from this example is as follows:

```
Price: 79600 Price: 275 Price: 48.95
```

You can see that the query is not evaluated until the results are enumerated, and so the change I made—introducing `Stadium` into the `Product` array—is reflected in the output.

■ **Tip** One interesting feature that arises from deferred LINQ extension methods is that queries are evaluated from scratch every time the results are enumerated, meaning that you can perform the query repeatedly as the source data for the changes and get results that reflect the current state of the source data.

By contrast, using any of the non-deferred extension methods causes a LINQ query to be performed immediately. Listing 4-31 shows the `SumProducts` action method I added to the `Home` controller.

Listing 4-31. An Immediately Executed LINQ Query in the HomeController.cs File

```
...
public ViewResult SumProducts() {

    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };

    var results = products.Sum(e => e.Price);

    products[2] = new Product { Name = "Stadium", Price = 79500M };

    return View("Result",
        (object)String.Format("Sum: {0:c}", results));
}
...
```

This example uses the `Sum` method, which is not deferred, and produces the following result:

```
Sum: $378.40
```

You can see that the `Stadium` item, with its much higher price, has not been included in the results—this is because the results from the `Sum` method are evaluated as soon as the method is called, rather than being deferred until the results are used.

Using Async Methods

One of the big recent additions to C# is improvements in the way that *asynchronous methods* are dealt with. Asynchronous methods go off and do work in the background and notify you when they are complete, allowing your code to take care of other business while the background work is performed. Asynchronous methods are an important tool in removing bottlenecks from code and allow applications to take advantage of multiple processors and processor cores to perform work in parallel.

C# and .NET have excellent support for asynchronous methods, but the code tends to be verbose and developers who are not used to parallel programming often get bogged down by the unusual syntax. As an example, Listing 4-32 shows an asynchronous method called `GetPageLength`, which I defined in a class called `MyAsyncMethods` and added to the `Models` folder in a class file called `MyAsyncMethods.cs`.

Listing 4-32. A Simple Asynchronous Method in the MyAsyncMethods.cs File

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public static Task<long?> GetPageLength() {

            HttpClient client = new HttpClient();

            var httpTask = client.GetAsync("http://apress.com");

            // we could do other things here while we are waiting
            // for the HTTP request to complete

            return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) => {
                return antecedent.Result.Content.Headers.ContentLength;
            });
        }
    }
}
```

■ **Caution** This example requires the `System.Net.Http` assembly, which I added to the project at the start of the chapter.

This is a simple method that uses a `System.Net.Http.HttpClient` object to request the contents of the Apress home page and returns its length. I have highlighted the part of the method that tends to cause confusion, which is an example of a *task continuation*.

.NET represents work that will be done asynchronously as a `Task`. `Task` objects are strongly typed based on the result that the background work produces. So, when I call the `HttpClient.GetAsync` method, what I get back is a `Task<HttpResponseMessage>`. This tells me that the request will be performed in the background and that the result of the request will be an `HttpResponseMessage` object.

■ **Tip** When I use words like *background*, I am skipping over a lot of detail in order to make the key points that are important to the world of MVC. The .NET support for asynchronous methods and parallel programming in general is excellent and I encourage you to learn more about it if you want to create truly high-performing applications that can take advantage of multicore and multiprocessor hardware. I come back to asynchronous methods for MVC in Chapter 19.

The part that most programmers get bogged down with is the *continuation*, which is the mechanism by which you specify what you want to happen when the background task is complete. In the example, I have used the `ContinueWith` method to process the `HttpResponseMessage` object I get from the `HttpClient.GetAsync` method, which I do using a lambda expression that returns the value of a property that returns the length of the content I get from the Apress Web server. Notice that I use the `return` keyword twice:

```
...
return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) => {
    return antecedent.Result.Content.Headers.ContentLength;
});
...
```

This is the part that makes heads hurt. The first use of the `return` keyword specifies that I am returning a `Task<HttpResponseMessage>` object, which, when the task is complete, will `return` the length of the `ContentLength` header. The `ContentLength` header returns a `long?` result (a nullable long value) and this means that the result of my `GetPageLength` method is `Task<long?>`, like this:

```
...
public static Task<long?> GetPageLength() {
...
```

Do not worry if this does not make sense—you are not alone in your confusion. And this is a simple example—complex asynchronous operations can chain large numbers of tasks together using the `ContinueWith` method, which creates code that can be hard to read and harder to maintain.

Applying the async and await Keywords

Microsoft introduced two keywords to C# that are specifically intended to simplify using asynchronous methods like `HttpClient.getAsync`. The keywords are `async` and `await` and you can see how I have used them to simplify my example method in Listing 4-33.

Listing 4-33. Using the Async and Await Keywords in the MyAsyncMethods.cs File

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public async static Task<long?> GetPageLength() {

            HttpClient client = new HttpClient();

            var httpMessage = await client.GetAsync("http://apress.com");

            // we could do other things here while we are waiting
            // for the HTTP request to complete
            return httpMessage.Content.Headers.ContentLength;
        }
    }
}
```

I used the `await` keyword when calling the asynchronous method. This tells the C# compiler that I want to wait for the result of the `Task` that the `GetAsync` method returns and then carry on executing other statements in the same method.

Applying the `await` keyword means I can treat the result from the `GetAsync` method as though it were a regular method and just assign the `HttpResponseMessage` object that it returns to a variable. And, even better, I can then use the `return` keyword in the normal way to produce a result from other method—in this case, the value of the `ContentLength` property. This is a much more natural technique and it means I do not have to worry about the `ContinueWith` method and multiple uses of the `return` keyword.

When you use the `await` keyword, you must also add the `async` keyword to the method signature, as I have done in the example. The method result type does not change—my example `GetPageLength` method still returns a `Task<long?>`. This is because the `await` and `async` are implemented using some clever compiler tricks, meaning that they allow a more natural syntax, but they do not change what is happening in the methods to which they are applied. Someone who is calling my `GetPageLength` method still has to deal with a `Task<long?>` result because there is still a background operation that produces a `nullable` long—although, of course, that programmer can also choose to use the `await` and `async` keywords as well.

■ **Note** You will have noticed that I did not provide an MVC example for you to test out the `async` and `await` keywords. This is because using asynchronous methods in MVC controllers requires a special technique, and I have a lot of information to present to you before I introduce it in Chapter 19.

Summary

In this chapter, I gave you an overview of the key C# language features that an effective MVC programmer needs to know. These features are combined in LINQ, which I use to query data throughout this book. As I said, I am a big fan of LINQ, and it plays an important role in MVC applications. I also showed you the `async` and `await` keywords, which make it easier to work with asynchronous methods—this is a topic that I return to in Chapter 19 when I show you an advanced technique for integrating asynchronous programming into your MVC controllers.

In the next chapter, I turn my attention to the Razor View Engine, which is the mechanism by which dynamic data is inserted into views.