CHAPTER 4

Essential C# Features

In this chapter, I describe C# features used in web application development that are not widely understood or that often cause confusion. This is not a book about C#, however, and so I provide only a brief example for each feature so that you can follow the examples in the rest of the book and take advantage of these features in your own projects. Table 4-1 summarizes this chapter.

Table 4-1. *Chapter Summary*

Problem	Solution	Listing
Avoid accessing properties on null references	Use the null conditional operator	5–8
Simplify C# properties	Use automatically implemented properties	9–11
Simplify string composition	Use string interpolation	12
Create an object and set its properties in a single step	Use an object or collection initializer	13–16
Test an object's type or characteristics	Use pattern matching	17–18
Add functionality to a class that cannot be modified	Use an extension method	19–26
Simplify the use of delegates and single-statement methods	Use a lambda expression	27–34
Use implicit typing	Use the `var` keyword	35
Create objects without defining a type	Use an anonymous type	36–37
Simplify the use of asynchronous methods	Use the `async` and `await` keywords	38–41
Get the name of a class method or property without defining a static string	Use a `nameof` expression	42–43

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_4

Preparing the Example Project

For this chapter, I created a new Visual Studio project called `LanguageFeatures` using the ASP.NET Core Web Application (.NET Core) template, as shown in Figure 4-1.

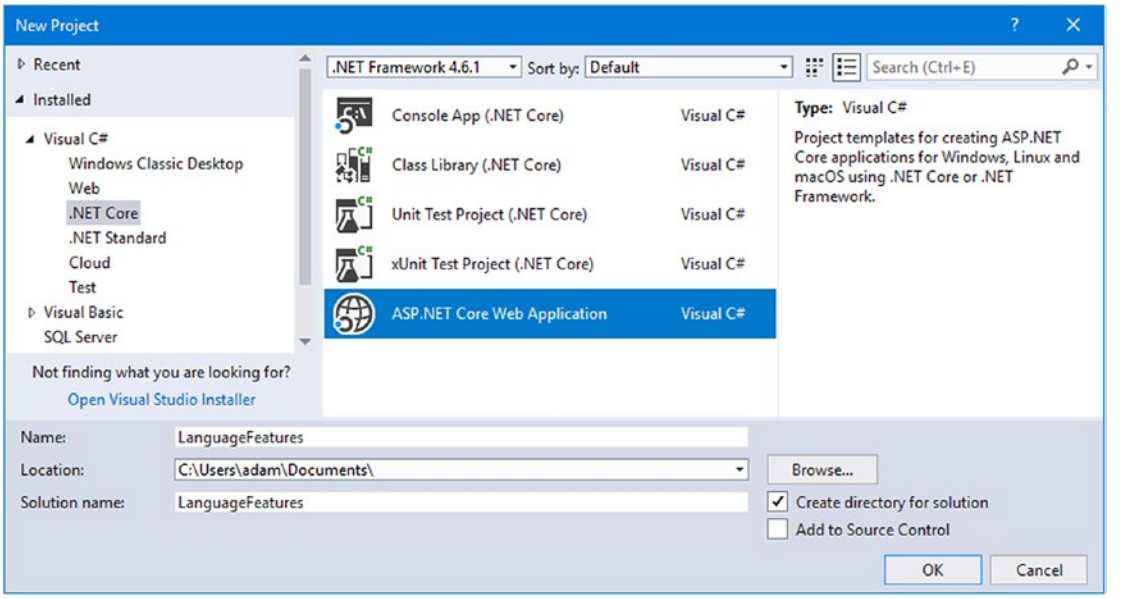

Figure 4-1. *Selecting the project type*

When presented with the different project configurations, I selected the Empty template, as shown in Figure 4-2. I selected .NET Core and ASP.NET Core 2.0 from the lists at the top of the dialog window, ensured that Authentication option was set to No Authentication and that the Enable Docker Support option was unchecked before clicking the OK button to create the project.

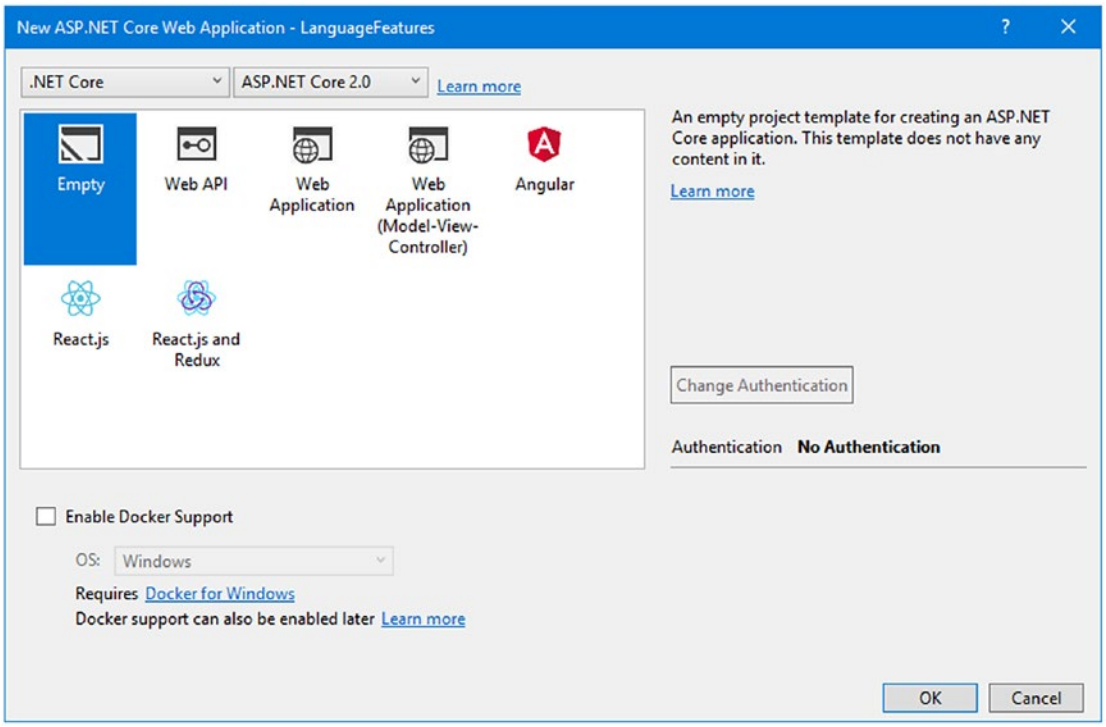

Figure 4-2. *Selecting the project template*

Enabling ASP.NET Core MVC

The Empty project template creates a project that contains a minimal ASP.NET Core configuration without any MVC support. This means that the placeholder content that is added by the Web Application (Model-View-Controller) template isn't present, but it also means that some extra steps are required to enable MVC so that features such as controllers and views work. In this section, I make the changes required to add enable an MVC setup in the project, but I won't get into the details of what each step does for the moment.

To enable the MVC framework, make the changes shown in Listing 4-1 to the `Startup` class.

Listing 4-1. Enabling MVC in the Startup.cs File in the LanguageFeatures Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace LanguageFeatures {

    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            //app.Run(async (context) => {
            //    await context.Response.WriteAsync("Hello World!");
            //});

            app.UseMvcWithDefaultRoute();
        }
    }
}
```

I explain how to configure ASP.NET Core MVC applications in Chapter 14, but the two statements added in Listing 4-1 provide a basic MVC setup using the default configuration and conventions.

Creating the MVC Application Components

Now that MVC is set up, I can add the MVC application components that I will use to demonstrate important C# language features.

Creating the Model

I started by creating a simple model class so that I can have some data to work with. I added a folder called `Models` and created a class file called `Product.cs` within it, which I used to define the class shown in Listing 4-2.

Listing 4-2. The Contents of the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };

            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

The `Products` class defines `Name` and `Price` properties, and there is a `static` method called `GetProducts` that returns a `Products` array. One of the elements contained in the array returned by the `GetProducts` method is set to `null`, which I will use to demonstrate some useful language features later in the chapter.

Creating the Controller and View

For the examples in this chapter, I use a simple controller to demonstrate different language features. I created a `Controllers` folder and added to it a class file called `HomeController.cs`, the contents of which are shown in Listing 4-3. When using the default MVC configuration, the `Home` controller is where MVC will send HTTP requests by default.

Listing 4-3. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View(new string[] { "C#", "Language", "Features" });
        }
    }
}
```

The `Index` action method tells MVC to render the default view and passes it an array of strings to be included in the HTML sent to the client. To create the corresponding view, I added a `Views/Home` folder (by creating a `Views` folder and then adding a `Home` folder within it) and added a view file called `Index.cshtml`, the contents of which are shown in Listing 4-4.

Listing 4-4. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model IEnumerable<string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Language Features</title>
</head>
<body>
    <ul>
        @foreach (string s in Model) {
            <li>@s</li>
        }
    </ul>
</body>
</html>
```

If you run the example application by selecting Start Debugging from the Debug menu, you will see the output shown in Figure 4-3.

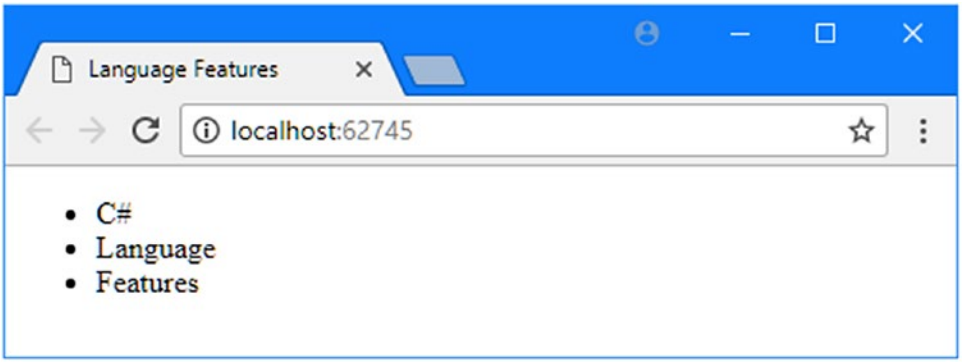

Figure 4-3. *Running the example application*

Since the output from all the examples in this chapter is text, I will show the messages displayed by the browser like this:

```
C#
Language
Features
```

Using the Null Conditional Operator

The null conditional operator allows for null values to be detected more elegantly. There can be a lot of checking for nulls in MVC development as you work out whether a request contains a specific header or value or whether the model contains a particular data item. Traditionally, dealing with nulls requires making an explicit check, and this can become tedious and error-prone when both an object and its properties have to be inspected. The null conditional operator makes this process simpler and more concise, as shown in Listing 4-5.

Listing 4-5. Detecting null Values in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()) {
                string name = p?.Name;
                decimal? price = p?.Price;
                results.Add(string.Format("Name: {0}, Price: {1}", name, price));
            }

            return View(results);
        }
    }
}
```

The static GetProducts method defined by the Product class returns an array of objects that I inspect in the controller's Index action method in order to get a list of the Name and Price values. The problem is that both the object in the array and the value of the properties could be null, which means I can't just refer to p.Name or p.Price within the foreach loop without causing a NullReferenceException. To avoid this, I used the null conditional operator, like this:

```
...
string name = p?.Name;
decimal? price = p?.Price;
...
```

The null conditional operator is a single question mark (the ? character). If p is null, then name will be set to null as well. If p is not null, then name will be set to the value of the Person.Name property. The Price property is subject to the same test. Notice that the variable you assign to when using the null conditional operator must be able to be assigned null, which is why the price variable is declared as a nullable decimal (decimal?).

Chaining the Null Conditional Operator

The null conditional operator can be chained to navigate through a hierarchy of objects, which is where it really becomes an effective tool for simplifying code and allowing safe navigation. In Listing 4-6, I have added a property to the `Product` class that nests references, creating a more complex object hierarchy.

Listing 4-6. Adding a Property in the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

Each `Product` object has a `Related` property that can refer to another `Product` object. In the `GetProducts` method, I set the `Related` property for the `Product` object that represents a kayak. Listing 4-7 shows how I can chain the null conditional operator to navigate through the object properties without causing an exception.

Listing 4-7. Detecting Nested null Values in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()) {
                string name = p?.Name;
                decimal? price = p?.Price;
                string relatedName = p?.Related?.Name;
```

```
                results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
                    name, price, relatedName));
            }

            return View(results);
        }
    }
}
```

The null conditional operator can be applied to each part of a chain of properties, like this:

```
...
string relatedName = p?.Related?.Name;
...
```

The result is that the relatedName variable will be null when p is null or when p.Related is null. Otherwise, the variable will be assigned the value of the p.Related.Name property. If you run the example, you will see the following output in the browser window:

```
Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related:
Name: , Price: , Related:
```

Combining the Conditional and Coalescing Operators

It can be useful to combine the null conditional operator (a single question mark) with the null coalescing operator (two question marks) to set a fallback value to present null values being used in the application, as shown in Listing 4-8.

Listing 4-8. Combining Null Operators in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()) {
                string name = p?.Name ?? "<No Name>";
                decimal? price = p?.Price ?? 0;
                string relatedName = p?.Related?.Name ?? "<None>";
                results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
                    name, price, relatedName));
            }
```

```
            return View(results);
        }
    }
}
```

The null conditional operator ensures that I don't get a `NullReferenceException` when navigating through the object properties, and the null coalescing operator ensures that I don't include `null` values in the results displayed in the browser. If you run the example, you will see the following results displayed in the browser window:

```
Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related: <None>
Name: <No Name>, Price: 0, Related: <None>
```

Using Automatically Implemented Properties

C# supports automatically implemented properties, and I used them when defining properties for the `Person` class in the previous section, like this:

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

This feature allows me to define properties without having to implement the `get` and `set` bodies. Using the auto-implemented property feature means that defining a property like this:

```
...
public string Name { get; set; }
...
```

is equivalent to the following code:

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

This type of feature is known as *syntactic sugar*, which means that it makes C# more pleasant to work with—in this case by eliminating redundant code that ends up being duplicated for every property—without substantially altering the way the language behaves. The term *sugar* may seem pejorative, but any enhancements that make code easier to write and maintain can be beneficial, especially in large and complex projects.

Using Auto-Implemented Property Initializers

Automatically implemented properties have been supported since C# 3.0. The latest version of C# supports initializers for automatically implemented properties, which allows an initial value to be set without having to use the constructor, as shown in Listing 4-9.

Listing 4-9. Using an Auto-Implemented Property Initializer in the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {
            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

Assigning a value to an auto-implemented property doesn't prevent the setter from being used to change the property later and just tidies up the code for simple types that ended up with a constructor that contained a list of property assignments to provide default values. In the example, the initializer assigns a value of `Watersports` to the `Category` property. The initial value can be changed, which I do when I create the `kayak` object and specify a value of `Water Craft` instead.

Creating Read-Only Automatically Implemented Properties

You can create a read-only property by using an initializer and omitting the `set` keyword from an auto-implemented property that has an initializer, as shown in Listing 4-10.

Listing 4-10. Creating a Read-Only Property in the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; } = true;

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

The `InStock` property is initialized to `true` and cannot be changed; however, the value can be assigned to in the type's constructor, as shown in Listing 4-11.

Listing 4-11. Assigning a Value to a Read-Only Property in the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public Product(bool stock = true) {
            InStock = stock;
        }

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; }
```

```
        public static Product[] GetProducts() {
            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };

            Product lifejacket = new Product(false) {
                Name = "Lifejacket",
                Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

The constructor allows the value for the read-only property to be specified as an argument and defaults to `true` if no value is provided. The property value cannot be changed once set by the constructor.

Using String Interpolation

The `string.Format` method is the traditional C# tool for composing strings that contain data values. Here is an example of this technique from the `Home` controller:

```
...
results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
                    name, price, relatedName));
...
```

C# also supports a different approach, known as *string interpolation*, that avoids the need to ensure that the `{0}` references in the string template match up with the variables specified as arguments. Instead, string interpolation uses the variable names directly, as shown in Listing 4-12.

Listing 4-12. Using String Interpolation in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()) {
                string name = p?.Name ?? "<No Name>";
```

```
                decimal? price = p?.Price ?? 0;
                string relatedName = p?.Related?.Name ?? "<None>";
                results.Add($"Name: {name}, Price: {price}, Related: {relatedName}");
            }

            return View(results);
        }
    }
}
```

Interpolated strings are prefixed with the $ character and contain *holes*, which are references to values contained within the { and } characters. When the string is evaluated, the holes are filled in with the current values of the variables or constants that are specified.

Visual Studio provides IntelliSense support for creating interpolated strings and offers a list of the available members when the { character is typed; this helps to minimize typos, and the result is a string format that is easier to understand.

■ **Tip** String interpolation supports all the format specifiers that are available with the `string.Format` method. The format specifiers are included as part of the hole, so `$"Price: {price:C2}"` would format the `price` value as a currency value with two decimal digits.

Using Object and Collection Initializers

When I create an object in the static `GetProducts` method of the `Product` class, I use an *object initializer*, which allows me to create an object and specify its property values in a single step, like this:

```
...
Product kayak = new Product {
    Name = "Kayak",
    Category = "Water Craft",
    Price = 275M
};
...
```

This is another syntactic sugar feature that makes C# easier to use. Without this feature, I would have to call the `Product` constructor and then use the newly created object to set each of the properties, like this:

```
...
Product kayak = new Product();
kayak.Name = "Kayak";
kayak.Category = "Water Craft";
kayak.Price = 275M;
...
```

A related feature is the *collection initializer*, which allows the creation of a collection and its contents to be specified in a single step. Without an initializer, creating a string array, for example, requires the size of the array and the array elements to be specified separately, as shown in Listing 4-13.

Listing 4-13. Initializing an Object in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            string[] names = new string[3];
            names[0] = "Bob";
            names[1] = "Joe";
            names[2] = "Alice";
            return View("Index", names);
        }
    }
}
```

Using a collection initializer allows the contents of the array to be specified as part of the construction, which implicitly provides the compiler with the size of the array, as shown in Listing 4-14.

Listing 4-14. Using a Collection Initializer in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View("Index", new string[] { "Bob", "Joe", "Alice" });
        }
    }
}
```

The array elements are specified between the { and } characters, which allows for a more concise definition of the collection and makes it possible to define a collection inline within a method call. The code in Listing 4-14 has the same effect as the code in Listing 4-13, and if you run the example application, you will see the following output in the browser window:

```
Bob
Joe
Alice
```

Using an Index Initializer

Recent versions of C# tidy up the way collections that use indexes, such as dictionaries, are initialized. Listing 4-15 shows the Index action rewritten to define a collection using the traditional C# approach to initializing a dictionary.

Listing 4-15. Initializing a Dictionary in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            Dictionary<string, Product> products = new Dictionary<string, Product> {
                { "Kayak", new Product { Name = "Kayak", Price = 275M } },
                { "Lifejacket",  new Product{ Name = "Lifejacket", Price = 48.95M } }
            };
            return View("Index", products.Keys);
        }
    }
}
```

The syntax for initializing this type of collection relies too much on the { and } characters, especially when the collection values are created using object initializers. The latest versions of C# support a more natural approach to initializing indexed collections that is consistent with the way that values are retrieved or modified once the collection has been initialized, as shown in Listing 4-16.

Listing 4-16. Using Collection Initializer Syntax in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            Dictionary<string, Product> products = new Dictionary<string, Product> {
                ["Kayak"] = new Product { Name = "Kayak", Price = 275M },
                ["Lifejacket"] = new Product { Name = "Lifejacket", Price = 48.95M }
            };

            return View("Index", products.Keys);
        }
    }
}
```

The effect is the same—to create a dictionary whose keys are `Kayak` and `Lifejacket` and whose values are `Product` objects—but the elements are created using the index notation that is used for other collection operations. If you run the application, you will see the following results in the browser:

```
Kayak
Lifejacket
```

Pattern Matching

One of the most useful recent additions to C# is support for pattern matching, which can be used to test that an object is of a specific type or has specific characteristics. This is another form is syntactic sugar, and it can dramatically simplify complex blocks of conditional statements. The `is` keyword is used to perform a type test, as demonstrated in Listing 4-17.

Listing 4-17. Performing a Type Test in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            object[] data = new object[] { 275M, 29.95M,
            "apple", "orange", 100, 10 };
            decimal total = 0;
            for (int i = 0; i < data.Length; i++) {
                if (data[i] is decimal d) {
                    total += d;
                }
            }

            return View("Index", new string[] { $"Total: {total:C2}" });
        }
    }
}
```

The `is` keyword performs a type check, and if a value is of the specified type, it will assign the value to a new variable, like this:

```
...
if (data[i] is decimal d) {
...
```

This expression will evaluate as `true` if the value stored in `data[i]` is a `decimal`. The value of `data[i]` will be assigned to the variable `d`, which allows it to be used in subsequent statements without needing to perform any type conversions. The `is` keyword will only match the specified type, which means that only two of the values in the data array will be processed (the other items in the array are `string` and `int` values). If you run the application, you will see the following output in the browser window:

```
Total: $304.95
```

Pattern Matching in Switch Statements

Pattern matching can also be used in `switch` statements, which support the `when` keyword for restricting when a value is matched by a `case` statement, as shown in Listing 4-18.

Listing 4-18. Pattern Matching in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            object[] data = new object[] { 275M, 29.95M,
                "apple", "orange", 100, 10 };
            decimal total = 0;
            for (int i = 0; i < data.Length; i++) {
                switch (data[i]) {
                    case decimal decimalValue:
                        total += decimalValue;
                        break;
                    case int intValue when intValue > 50:
                        total += intValue;
                        break;
                }
            }

            return View("Index", new string[] { $"Total: {total:C2}" });
        }
    }
}
```

To match any value of a specific type, use the type and variable name in the `case` statement, like this:

```
...
case decimal decimalValue:
...
```

This case statement matches any decimal value and assigns it to a new variable called decimalValue. To be more selective, the when keyword can be included, like this:

```
...
case int intValue when intValue > 50:
...
```

This case statement matches int values and assigns them to a variable called intValue, but only when the value is greater than 50. If you run the application, you will see the following output in the browser window:

```
Total: $404.95
```

Using Extension Methods

Extension methods are a convenient way of adding methods to classes that you do not own and cannot modify directly. Listing 4-19 shows the definition of the ShoppingCart class, which I added to the Models folder in a file called ShoppingCart.cs and which represents a collection of Product objects.

Listing 4-19. The Contents of the ShoppingCart.cs File in the Models Folder

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

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

This is a simple class that acts as a wrapper around a sequence 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 4-20 shows the MyExtensionMethods class that I added to the Models folder in the MyExtensionMethods.cs file.

Listing 4-20. The Contents of the MyExtensionMethods.cs File in the Models Folder

```
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 ?? 0;
            }
            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 class that the extension method has been applied to by using the cartParam parameter. My method enumerates the Product objects in ShoppingCart and returns the sum of the Product.Price property values. Listing 4-21 shows how I apply the extension method in the Home controller's action method.

■ **Note** Extension methods do not let you break through the access rules that classes define for 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.

Listing 4-21. Applying an Extension Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            ShoppingCart cart
            = new ShoppingCart { Products = Product.GetProducts() };
            decimal cartTotal = cart.TotalPrices();
            return View("Index", new string[] { $"Total: {cartTotal:C2}" });
        }
    }
}
```

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. If you run the application, you will see the following output in the browser window:

```
Total: $323.95
```

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 the classes that implement the interface. Listing 4-22 shows the ShoppingCart class updated to implement the IEnumerable<Product> interface.

Listing 4-22. Implementing an Interface in the ShoppingCart.cs File in the Models Folder

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

namespace LanguageFeatures.Models {

    public class ShoppingCart : IEnumerable<Product> {
        public IEnumerable<Product> Products { get; set; }

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

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

I can now update the extension method so that it deals with `IEnumerable<Product>`, as shown in Listing 4-23.

Listing 4-23. Updating an Extension Method in the MyExtensionMethods.cs File in the Models Folder

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumerable<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            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 change to using 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 `Product` objects, as shown in Listing 4-24.

Listing 4-24. Applying an Extension Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

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

```
        public ViewResult Index() {

            ShoppingCart cart
                = new ShoppingCart { Products = Product.GetProducts() };

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M}
            };

            decimal cartTotal = cart.TotalPrices();
            decimal arrayTotal = productArray.TotalPrices();

            return View("Index", new string[] {
                $"Cart Total: {cartTotal:C2}",
                $"Array Total: {arrayTotal:C2}" });
        }
    }
}
```

If you start the project, 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: $323.95
Array Total: $323.95
```

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 4-25 demonstrates such a method, which I have added to the `MyExtensionMethods` class.

Listing 4-25. A Filtering Extension Method in the MyExtensionMethods.cs File in the Controllers Folder

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

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

```
        public static IEnumerable<Product> FilterByPrice(
                this IEnumerable<Product> productEnum, decimal minimumPrice) {

            foreach (Product prod in productEnum) {
                if ((prod?.Price ?? 0) >= minimumPrice) {
                    yield return prod;
                }
            }
        }
    }
}
```

This extension method, called `FilterByPrice`, takes an additional parameter that allows me to filter products so that `Product` objects whose `Price` property matches or exceeds the parameter are returned in the result. Listing 4-26 shows this method being used.

Listing 4-26. Using the Filtering Extension Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            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}
            };

            decimal arrayTotal = productArray.FilterByPrice(20).TotalPrices();

            return View("Index", new string[] { $"Array Total: {arrayTotal:C2}" });
        }
    }
}
```

When I call the `FilterByPrice` method on the array of `Product` objects, only those that cost more than $20 are received by the `TotalPrices` method and used to calculate the total. If you run the application, you will see the following output in the browser window:

```
Total: $358.90
```

Using Lambda Expressions

Lambda expressions are a feature that causes a lot of confusion, not least because the feature they simplify is also confusing. To understand the problem that is being solved, consider the `FilterByPrice` extension method that I defined in the previous section. This method is written so that it can filter `Product` objects by price, which means that if I want to filter by name, I have to create a second method, like the one shown in Listing 4-27.

Listing 4-27. Adding a Filter Method in the MyExtensionMethods.cs File in the Models Folder

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

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

        public static IEnumerable<Product> FilterByPrice(
                this IEnumerable<Product> productEnum, decimal minimumPrice) {

            foreach (Product prod in productEnum) {
                if ((prod?.Price ?? 0) >= minimumPrice) {
                    yield return prod;
                }
            }
        }

        public static IEnumerable<Product> FilterByName(
                this IEnumerable<Product> productEnum, char firstLetter) {

            foreach (Product prod in productEnum) {
                if (prod?.Name?[0] == firstLetter) {
                    yield return prod;
                }
            }
        }
    }
}
```

Listing 4-28 shows the use of both filter methods applied in the controller to create two different totals.

Listing 4-28. Using Two Filter Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            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}
            };

            decimal priceFilterTotal = productArray.FilterByPrice(20).TotalPrices();
            decimal nameFilterTotal = productArray.FilterByName('S').TotalPrices();

            return View("Index", new string[] {
                $"Price Total: {priceFilterTotal:C2}",
                $"Name Total: {nameFilterTotal:C2}" });
        }
    }
}
```

The first filter selects all of the products with a price of $20 or more, and the second filter selects products whose name starts with the letter *S*. You will see the following output in the browser window if you run the example application:

```
Price Total: $358.90
Name Total: $19.50
```

Defining Functions

I can repeat this process indefinitely to create filter methods for every property and every combination of properties that I am interested in. A more elegant approach is to separate out the code that processes the enumeration from the selection criteria. C# makes this easy by allowing functions to be passed around as objects. Listing 4-29 shows a single extension method that filters an enumeration of `Product` objects but that delegates the decision about which ones are included in the results to a separate function.

Listing 4-29. Creating a General Filter Method in the MyExtensionMethods.cs File in the Models Folder

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

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

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

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

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

The second argument to the `Filter` method is a function that accepts a `Product` object and that returns a `bool` value. The `Filter` method calls the function for each `Product` object and includes it in the result if the function returns `true`. To use the `Filter` method, I can specify a method or create a stand-alone function, as shown in Listing 4-30.

Listing 4-30. Using a Function to Filter Objects in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        bool FilterByPrice(Product p) {
            return (p?.Price ?? 0) >= 20;
        }

        public ViewResult Index() {

            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}
        };

        Func<Product, bool> nameFilter = delegate (Product prod) {
            return prod?.Name?[0] == 'S';
        };

        decimal priceFilterTotal = productArray
            .Filter(FilterByPrice)
            .TotalPrices();
        decimal nameFilterTotal = productArray
            .Filter(nameFilter)
            .TotalPrices();

        return View("Index", new string[] {
            $"Price Total: {priceFilterTotal:C2}",
            $"Name Total: {nameFilterTotal:C2}" });
    }
  }
}
```

Neither approach is ideal. Defining methods like `FilterByPrice` clutters up a class definition. Creating a `Func<Product, bool>` object avoids this problem but uses an awkward syntax that is hard to read and hard to maintain. It is this issue that lambda expressions address by allowing functions to be defined in a more elegant and expressive way, as shown in Listing 4-31.

Listing 4-31. Using Lambda Expression in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

            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}
            };

            decimal priceFilterTotal = productArray
                .Filter(p => (p?.Price ?? 0) >= 20)
                .TotalPrices();
            decimal nameFilterTotal = productArray
```

```
                .Filter(p => p?.Name?[0] == 'S')
                .TotalPrices();

            return View("Index", new string[] {
                $"Price Total: {priceFilterTotal:C2}",
                $"Name Total: {nameFilterTotal:C2}" });
        }
    }
}
```

The lambda expressions are shown in bold. The parameters are expressed without specifying a type, which will be inferred automatically. The => characters are read aloud as "goes to" and link the parameter to the result of the lambda expression. In my examples, a `Product` parameter called `p` goes to a `bool` result, which will be `true` if the `Price` property is equal or greater than 20 in the first expression or if the `Name` property starts with *S* in the second expression. This code works in the same way as the separate method and the function delegate but is more concise and is—for most people—easier to read.

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
```

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 Lambda Expression Methods and Properties

Lambda expressions can be used to implement constructors, methods, and properties. In MVC development, especially when writing controllers, you will often end up with methods that contain a single statement that selects the data to display and the view to render. In Listing 4-32, I have rewritten the `Index` action method so that it follows this common pattern.

Listing 4-32. Creating a Common Action Pattern in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View(Product.GetProducts().Select(p => p?.Name));
        }
    }
}
```

The action method gets a collection of `Product` objects from the static `Product.GetProducts` method and uses LINQ to project the values of the `Name` properties, which are then used as the view model for the default view. If you run the application, you will see the following output displayed in the browser window:

```
Kayak
Lifejacket
```

There will be an empty list item in the browser window as well because the `GetProducts` method includes a `null` reference in its results, but that doesn't matter for this section of the chapter.

When a constructor or method body consists of a single statement, it can be rewritten as a lambda expression, as shown in Listing 4-33.

Listing 4-33. An Action Method Expressed as a Lambda Expression in the HomeController.cs File

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() =>
        View(Product.GetProducts().Select(p => p?.Name));
    }
}
```

Lambda expressions for methods omit the `return` keyword and use => (goes to) to associate the method signature (including its arguments) with its implementation. The `Index` method shown in Listing 4-33 works in the same way as the one shown in Listing 4-32 but is expressed more concisely. The same basic approach can also be used to define properties. Listing 4-34 shows the addition of a property that uses a lambda express to the `Product` class.

Listing 4-34. Expressing a Property as a Lambda Expression in the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public Product(bool stock = true) {
            InStock = stock;
        }

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; }
        public bool NameBeginsWithS => Name?[0] == 'S';

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };

            Product lifejacket = new Product(false) {
                Name = "Lifejacket",
                Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

Using Type Inference and Anonymous Types

The var keyword allows you to define a local variable without explicitly specifying the variable type, as demonstrated by Listing 4-35. This is called *type inference* or *implicit typing*.

Listing 4-35. Using Type Inference in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

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

```
        public ViewResult Index() {
            var names = new [] { "Kayak", "Lifejacket", "Soccer ball" };
            return View(names);
        }
    }
}
```

It is not that the names variable does not have a type; instead, I am asking the compiler to infer the type from the code. The compiler examines the array declaration and works out that it is a string array. Running the example produces the following output:

```
Kayak
Lifejacket
Soccer ball
```

Using Anonymous Types

By combining object initializers and type inference, I can create simple view model objects that are useful for transferring data between a controller and a view without having to define a class or struct, as shown in Listing 4-36.

Listing 4-36. Creating an Anonymous Type in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.95M },
                new { Name = "Soccer ball", Price = 19.50M },
                new { Name = "Corner flag", Price = 34.95M }
            };

            return View(products.Select(p => p.Name));
        }
    }
}
```

Each of the objects in the products array 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. If you run the example, you will see the following output in the browser window:

```
Kayak
Lifejacket
Soccer ball
Corner flag
```

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 that all the objects in the products array will have the same type because they define the same properties.

■ **Tip** I have to use the var keyword to define the array of anonymously typed objects because the type isn't created until the code is compiled and so I don't know the name of the type to use. The elements in an array of anonymously typed objects must all define the same properties; otherwise, the compiler can't work out what the array type should be.

To demonstrate this, I have changed the output from the example in Listing 4-37 so that it shows the type name rather than the value of the Name property.

Listing 4-37. Displaying the Type Name in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.95M },
                new { Name = "Soccer ball", Price = 19.50M },
                new { Name = "Corner flag", Price = 34.95M }
            };

            return View(products.Select(p => p.GetType().Name));
        }
    }
}
```

All the objects in the array have been assigned the same type, which you can see if you run the example. The type name isn't user-friendly but isn't intended to be used directly, and you may see a different name than the one shown in the following output:

```
<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2
```

Using Asynchronous Methods

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.

In MVC, asynchronous methods can be used to improve the overall performance of an application by allowing the server more flexibility in the way that requests are scheduled and executed. Two C# keywords—`async` and `await`—are used to perform work asynchronously.

To prepare for this section, I need to add a new .NET assembly to the example project so that I can make asynchronous HTTP requests. Right-click the LanguageFeatures project item in the Solution Explorer, select Edit LanguageFeatures.csproj from the pop-up menu, and add the element shown in Listing 4-38.

Listing 4-38. Adding a Package in the LanguageFeatures.csproj File in the LanguageFeatures Folder

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="System.Net.Http" Version="4.3.2" />
  </ItemGroup>

</Project>
```

When you save the file, Visual Studio will download the `System.Net.Http` assembly and add it to the project. I describe this process in more detail in Chapter 6.

Working with Tasks Directly

C# and .NET have excellent support for asynchronous methods, but the code has tended to be verbose, and developers who are not used to parallel programming often get bogged down by the unusual syntax. As an example, Listing 4-39 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 4-39. The Contents of the MyAsyncMethods.cs File in the Models Folder

```
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");

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

This method uses a `System.Net.Http.HttpClient` object to request the contents of the Apress home page and returns its length. .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 to make just the key points that are important to the world of MVC. The .NET support for asynchronous methods and parallel programming 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. You will see how MVC makes it easy to create asynchronous web applications throughout this book as I introduce different features.

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 contains the length of the content I get from the Apress web server. Here is the continuation code:

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

Notice that I use the `return` keyword twice. This is the part that causes confusion. 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. It is for this reason that Microsoft added keywords to C# to simplify asynchronous methods.

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 4-40.

Listing 4-40. Using the async and await Keywords in the MyAsyncMethods.cs File in the Models Folder

```
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");

            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. Even better, I can then use the `return` keyword in the normal way to produce a result from another 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 `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.

This pattern follows through into the MVC controller, which makes it easy to write asynchronous action methods, as shown in Listing 4-41.

Listing 4-41. An Asynchronous Action Methods in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public async Task<ViewResult> Index() {
            long? length = await MyAsyncMethods.GetPageLength();
            return View(new string[] { $"Length: {length}" });
        }
    }
}
```

I have changed the result of the `Index` action method to `Task<ViewResult>`, which tells MVC that the action method will return a `Task` that will produce a `ViewResult` object when it completes, which will provide details of the view that should be rendered and the data that it requires. I have added the `async` keyword to the method's definition, which allows me to use the `await` keyword when calling the `MyAsyncMethods.GetPathLength` method. MVC and .NET take care of dealing with the continuations, and the result is asynchronous code that is easy to write, easy to read, and easy to maintain. If you run the application, you will see output similar to the following (although with a different length since the content of the Apress web site changes often):

```
Length: 54576
```

Getting Names

There are many tasks in web application development in which you need to refer to the name of an argument, variable, method, or class. Common examples include when you throw an exception or create a validation error when processing input from the user. The traditional approach has been to use a string value hard-coded with the name, as shown in Listing 4-42.

Listing 4-42. Hard-Coding a Name in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {

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

            return View(products.Select(p => $"Name: {p.Name}, Price: {p.Price}"));
        }
    }
}
```

The call to the LINQ `Select` method generates a sequence of strings, each of which contains a hard-coded reference to the `Name` and `Price` properties. Running the application produces the following output in the browser window:

```
Name: Kayak, Price: 275
Name: Lifejacket, Price: 48.95
Name: Soccer ball, Price: 19.50
Name: Corner flag, Price: 34.95
```

The problem with this approach is that it is prone to errors, either because the name was mistyped or the code was refactored and the name in the string isn't correctly updated. The result can be misleading, which can be especially problematic for messages that are displayed to the user. C# supports the `nameof` expression, in which the compiler takes responsibility for producing a name string, as shown in Listing 4-43.

Listing 4-43. Using nameof Expressions in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

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

```
        public ViewResult Index() {

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

            return View(products.Select(p =>
                $"{nameof(p.Name)}: {p.Name}, {nameof(p.Price)}: {p.Price}"));
        }
    }
}
```

The compiler processes a reference such as `p.Name` so that only the last part is included in the string, producing the same output as in previous examples. Visual Studio includes IntelliSense support for `nameof` expressions, so you will be prompted to select references, and expressions will be correctly updated when you refactor code. Since the compiler is responsible for dealing with `nameof`, using an invalid reference causes a compiler error, which prevents incorrect or outdated references from escaping notice.

Summary

In this chapter, I gave you an overview of the key C# language features that an effective MVC programmer needs to know. C# is a sufficiently flexible language that there are usually different ways to approach any problem, but these are the features that you will encounter most often during web application development and see throughout the examples in this book. In the next chapter, I introduce the Razor view engine and explain how it is used to generate dynamic content in MVC web applications.