

## CHAPTER 5



# 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, 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 5-1 summarizes this chapter.

**Table 5-1.** Chapter Summary

Problem	Solution	Listing
Managing null values	Use the null conditional and null coalescing operators	7, 10
Creating properties with getters and setters	Define automatically implemented properties	11–13
Mixing static and dynamic values in strings	Use string interpolation	14
Initializing and populate objects	Use the object and collection initializers	15–18
Assigning a value for specific types	Use pattern matching	19, 20
Extending the functionality of a class without modifying it	Define an extension method	21–28
Expressing functions and methods concisely	Use lambda expressions	29–36
Defining a variable without explicitly declaring its type	Use the var keyword	37–39
Modifying an interface without requiring changes in its implementation classes	Define a default implementation	40–44
Performing work asynchronously	Use tasks or the async/await keywords	45–47
Producing a sequence of values over time	Use an asynchronous enumerable	48–51
Getting the name of a class or member	Use a nameof expression	52, 53

## Preparing for This Chapter

To create the example project for this chapter, open a new PowerShell command prompt and run the commands shown in Listing 5-1. If you are using Visual Studio and prefer not to use the command line, you can create the project using the process described in Chapter 4.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

**Listing 5-1.** Creating the Example Project

---

```
dotnet new globaljson --sdk-version 3.1.101 --output LanguageFeatures
dotnet new web --no-https --output LanguageFeatures --framework netcoreapp3.1
dotnet new sln -o LanguageFeatures

dotnet sln LanguageFeatures add LanguageFeatures
```

---

## Opening the Project

If you are using Visual Studio, select File ► Open ► Project/Solution, select the `LanguageFeatures.sln` file in the `LanguageFeatures` folder, and click the Open button to open the solution file and the project it references. If you are using Visual Studio Code, select File ► Open Folder, navigate to the `LanguageFeatures` folder, and click the Select Folder button.

## Enabling the MVC Framework

The web project template creates a project that contains a minimal ASP.NET Core configuration. This means the placeholder content that is added by the `mvc` template used in Chapter 3 is not available and that extra steps are required to reach the point where the application can produce useful output. In this section, I make the changes required to set up the MVC Framework, which is one of the application frameworks supported by ASP.NET Core, as I explained in Chapter 1. First, to enable the MVC framework, make the changes shown in Listing 5-2 to the `Startup` class.

**Listing 5-2.** 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;
using Microsoft.Extensions.Hosting;

namespace LanguageFeatures {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
        }

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

            app.UseRouting();

            app.UseEndpoints(endpoints => {
                //endpoints.MapGet("/", async context => {
                //    await context.Response.WriteAsync("Hello World!");
                //});
                endpoints.MapDefaultControllerRoute();
            });
        }
    }
}
```

I explain how to configure ASP.NET Core applications in Part 2, but the two statements added in Listing 5-2 provide a basic MVC framework setup using a default configuration.

## Creating the Application Components

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

### Creating the Data 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 5-3.

**Listing 5-3.** 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 `Product` class defines `Name` and `Price` properties, and there is a static method called `GetProducts` that returns a `Product` 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 class 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 5-4.

**Listing 5-4.** The Contents of the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

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

The Index action method tells ASP.NET Core to render the default view and provides it with an array of strings as its view model, which will be included in the HTML sent to the client. To create the view, I added a Views/Home folder (by creating a Views folder and then adding a Home folder within it) and added a Razor View called Index.cshtml, the contents of which are shown in Listing 5-5.

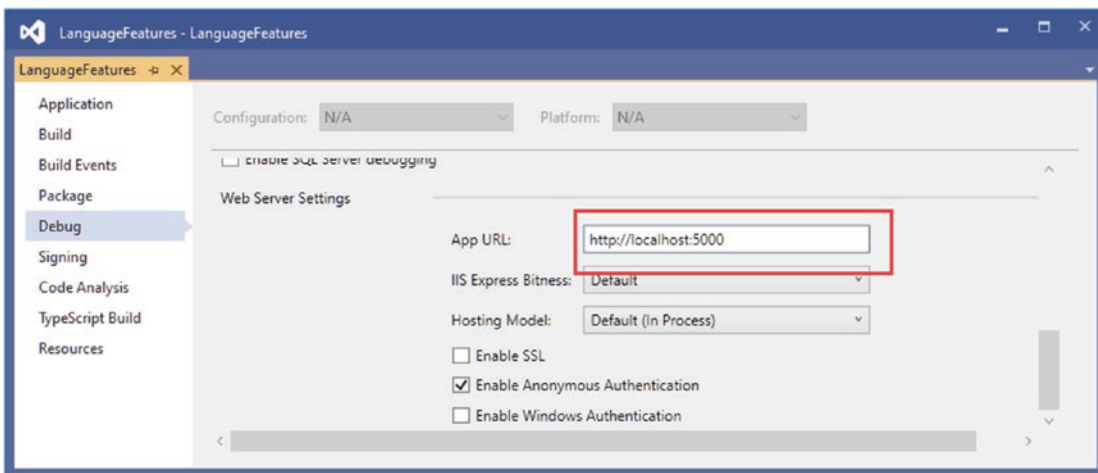
**Listing 5-5.** The Contents of the Index.cshtml File in the Views/Home Folder

```
@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>
```

## Selecting the HTTP Port

If you are using Visual Studio, select Project ► LanguageFeatures Properties, select the Debug section, and change the HTTP port to 5000 in the App URL field, as shown in Figure 5-1. Select File ► Save All to save the new port. (This change is not required if you are using Visual Studio Code.)



**Figure 5-1.** Setting the HTTP port

## Running the Example Application

Start ASP.NET Core by selecting Start Without Debugging (Visual Studio) or Run Without Debugging (Visual Studio Code) from the Debug menu or by running the command shown in Listing 5-6 in the LanguageFeatures folder.

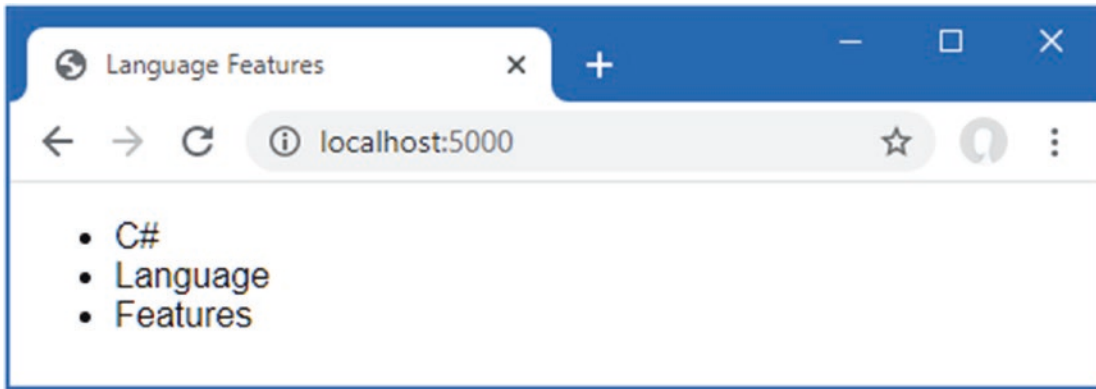
**Listing 5-6.** Running the Example Application

---

```
dotnet run
```

---

Request `http://localhost:5000`, and you will see the output shown in Figure 5-2.

**Figure 5-2.** 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 ASP.NET Core development as you work out whether a request contains a specific header or value or whether the model contains a specific data item. Traditionally, dealing with null values requires making an explicit check, which can become tedious and error-prone when both an object and its properties need to be inspected. The null conditional operator makes this process simpler and more concise, as shown in Listing 5-7.

**Listing 5-7.** 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 IActionResult 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 `Index` action method 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 becomes an effective tool for simplifying code and allowing safe navigation. In Listing 5-8, I have added a property to the `Product` class that creates a more complex object hierarchy.

**Listing 5-8.** 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 5-9 shows how I can chain the null conditional operator to navigate through the object properties without causing an exception.

**Listing 5-9.** 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 IActionResult 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. Restart ASP.NET Core and request <http://localhost:5000>, and 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 prevent null values being used in the application, as shown in Listing 5-10.

**Listing 5-10.** 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>

```

---

## NULLABLE AND NON-NULLABLE REFERENCE TYPES

Encountering unexpected null values is one of the most common causes of bugs. By default, C# treats null as a valid value for all types, which means that I can assign null to a string variable, like this:

```

...
string product = null;
...

```

It is the responsibility of the code that uses the variable to check for null values, which can be especially problematic when the same variable is in multiple places. It is easy to omit one of the checks or assume that a value won't be null, producing an error at runtime.

Nullable reference types shift responsibility for null checking to the code that assigns the value to a variable. When the nullable reference feature is enabled, regular reference types cannot be assigned null values, such as assigning null to a string, for example. Instead, nullable reference types must be used if null values are possible, like this:

```

...
string product = null; // compiler error - this is a non-nullable type
string? product = null; // no error - this is a nullable type
...

```

The `string?` type is nullable, while `string` is not, which means that the code that consumes a variable doesn't have to worry about null values unless it is dealing with a nullable type. To enable nullable reference types, an element must be added to the csproj file, like this:

```

...
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

```



```
</Project>
...
```

If you are using Visual Studio, you can open the project file by right-clicking the project item in the Solution Explorer and selecting Edit Project File from the popup menu.

I like this feature, but it is not yet used widely enough for me to use it in this book, especially since it can make some complex topics even more difficult to follow. But, once the rest of .NET Core catches up with this feature, I expect it to be embraced in ASP.NET Core by default, and you can expect to see nullable reference types used in future editions of this book.

---

## 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 I can define 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 use—in this case by eliminating redundant code that ends up being duplicated for every property—without substantially altering the way that 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 5-11.

**Listing 5-11.** 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 5-12.

**Listing 5-12.** 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 5-13.

**Listing 5-13.** 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 5-14.

**Listing 5-14.** 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 IActionResult 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.

---

■ **Tip** String interpolation supports all the format specifiers that are available with the `string.Format` method. The format specifics 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 5-15.

**Listing 5-15.** 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 IActionResult 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 5-16.

**Listing 5-16.** 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 IActionResult 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 5-16 has the same effect as the code in Listing 5-15. Restart ASP.NET Core and request <http://localhost:5000>, and 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 5-17 shows the Index action rewritten to define a collection using the traditional C# approach to initializing a dictionary.

**Listing 5-17.** 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 IActionResult 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 5-18.

**Listing 5-18.** 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 IActionResult Index() {
            Dictionary<string, Product> products = new Dictionary<string, Product> {
                ["Kayak"] = new Product { Name = "Kayak", Price = 275M },
            };
            return View("Index", products.Keys);
        }
    }
}

```

```

        ["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. Restart ASP.NET Core and request `http://localhost:5000`, and 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 of 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 5-19.

**Listing 5-19.** 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 IActionResult 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, will assign the value to a new variable, like this:

```

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

```

This expression evaluates 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 match only 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 5-20.

**Listing 5-20.** 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 IActionResult 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. Restart ASP.NET Core and request `http://localhost:5000`, and 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 cannot modify directly, typically because they are provided by Microsoft or a third-party package. Listing 5-21 shows the definition of the `ShoppingCart` class, which I added to the `Models` folder in a class file called `ShoppingCart.cs` file and which represents a collection of `Product` objects.

**Listing 5-21.** 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 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.

Add a class file named `MyExtensionMethods.cs` in the `Models` folder and use it to define the class shown in Listing 5-22.

**Listing 5-22.** 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;
        }
    }
}
```

Extension methods are defined in static classes within the same namespace as the class the extension methods applies to. In this case, the static `MyExtensionMethods` class is in the `LanguageFeatures.Models` namespace, which means that it can contain extension methods for classes in that namespace.

Extension methods are also static, and Listing 5-22 defines a single extension method named `TotalPrices`. 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. This extension method enumerates the `Product` objects in the `ShoppingCart` and returns the sum of the `Product.Price` property values. Listing 5-23 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 5-23.** 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 IActionResult 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. Restart ASP.NET Core and request `http://localhost:5000`, which will produce the following output in the browser window:

---

Total: \$323.95

---

## Applying Extension Methods to an Interface

Extension methods can also be applied to an interface, which allows me to call the extension method on all the classes that implement the interface. Listing 5-24 shows the `ShoppingCart` class updated to implement the `IEnumerable<Product>` interface.

**Listing 5-24.** 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 5-25.

**Listing 5-25.** 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 5-26.

**Listing 5-26.** 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 IActionResult 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}" });
        }
    }
}

```

Restart ASP.NET Core and request `http://localhost:5000`, which will produce the following output in the browser, demonstrating 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 5-27 demonstrates such a method, which I have added to the `MyExtensionMethods` class.

**Listing 5-27.** A Filtering 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;
        }

        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 5-28 shows this method being used.

**Listing 5-28.** 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 ActionResult 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 I must create a second method I want to filter by name, as shown in Listing 5-29.

**Listing 5-29.** 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 5-30 shows the use of both filter methods applied in the controller to create two different totals.

**Listing 5-30.** 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 IActionResult 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 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 the code that processes the enumeration from the selection criteria. C# makes this easy by allowing functions to be passed around as objects. Listing 5-31 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 5-31.** 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 5-32.

**Listing 5-32.** 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 IActionResult 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 5-33.

**Listing 5-33.** Using a 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 IActionResult 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 throughout this book.

## Using Lambda Expression Methods and Properties

Lambda expressions can be used to implement constructors, methods, and properties. In ASP.NET Core development, 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 5-34, I have rewritten the `Index` action method so that it follows this common pattern.

**Listing 5-34.** 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 IActionResult 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 5-35.

**Listing 5-35.** A Lambda Action Method 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 IActionResult 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 5-35 works in the same way as the one shown in Listing 5-34 but is expressed more concisely. The same basic approach can also be used to define properties. Listing 5-36 shows the addition of a property that uses a lambda expression to the `Product` class.

**Listing 5-36.** A Lambda 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 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 5-37. This is called *type inference*, or *implicit typing*.

**Listing 5-37.** 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 IActionResult 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 5-38.

**Listing 5-38.** 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 IActionResult 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. Restart ASP.NET Core and request `http://localhost:5000`, and 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, 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 5-39 so that it shows the type name rather than the value of the `Name` property.

**Listing 5-39.** 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 IActionResult 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 Default Implementations in Interfaces

C# 8.0 introduces the ability to define default implementations for properties and methods defined by interfaces. This may seem like an odd feature because interfaces are intended to be a description of features without specifying an implementation, but this addition to C# makes it possible to update interfaces without breaking the existing implementations of them.

Add a class file named `IProductSelection.cs` to the `Models` folder and use it to define the interface shown in Listing 5-40.

**Listing 5-40.** The Contents of the `IProductSelection.cs` File in the `Models` Folder

```

using System.Collections.Generic;

namespace LanguageFeatures.Models {
    public interface IProductSelection {

        IEnumerable<Product> Products { get; }
    }
}

```

Update the `ShoppingCart` class to implement the new interface, as shown in Listing 5-41.

**Listing 5-41.** 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 : IProductSelection {
        private List<Product> products = new List<Product>();

        public ShoppingCart(params Product[] prods) {
            products.AddRange(prods);
        }

        public IEnumerable<Product> Products { get => products; }
    }
}

```

Listing 5-42 updates the Home controller so that it uses the ShoppingCart class.

**Listing 5-42.** Using an Interface 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 IActionResult Index() {
            IProductSelection cart = new ShoppingCart(
                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 }
            );
            return View(cart.Products.Select(p => p.Name));
        }
    }
}
```

This is the familiar use of an interface, and if you restart ASP.NET Core and request `http://localhost:5000`, you will see the following output in the browser:

---

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

---

If I want to add a new feature to the interface, I must locate and update all the classes that implement it, which can be difficult, especially if an interface is used by other development teams in their projects. This is where the default implementation feature can be used, allowing new features to be added to an interface, as shown in Listing 5-43.

**Listing 5-43.** Adding a Feature in the IProductSelection.cs File in the Models Folder

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

namespace LanguageFeatures.Models {
    public interface IProductSelection {

        IEnumerable<Product> Products { get; }

        IEnumerable<string> Names => Products.Select(p => p.Name);
    }
}
```

The listing defines a Names property and provides a default implementation, which means that consumers of the IProductSelection interface can use the Total property even if it isn't defined by implementation classes, as shown in Listing 5-44.

**Listing 5-44.** Using a Default Implementation 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 IActionResult Index() {

            IProductSelection cart = new ShoppingCart(
                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 }
            );
            return View(cart.Names);
        }
    }
}
```

The ShoppingCart class has not been modified, but the Index method is able to use the default implementation of the Names property. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser:

---

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

---

## Using Asynchronous Methods

Asynchronous methods perform 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 ASP.NET Core, 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.

## 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. To create an example, add a class file called `MyAsyncMethods.cs` to the Models folder and add the code shown in Listing 5-45.

**Listing 5-45.** 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 ASP.NET Core. 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 ASP.NET Core 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 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 with 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 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 5-46.



**Listing 5-46.** 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 controller, which makes it easy to write asynchronous action methods, as shown in Listing 5-47.

---

■ **Note** You can also use the `async` and `await` keywords in lambda expressions, which I demonstrate in later chapters.

---

**Listing 5-47.** 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 declares 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. .NET takes care of dealing with the continuations, and the result is asynchronous code that is easy to write, easy to read, and easy to maintain. Restart ASP.NET Core and request `http://localhost:5000`, and you will see output similar to the following (although with a different length since the content of the Apress web site changes often):

---

Length: 101868

---

## Using an Asynchronous Enumerable

An asynchronous enumerable describes a sequence of values that will be generated over time. To demonstrate the issue that this feature addresses, Listing 5-48 adds a method to the `MyAsyncMethods` class.

**Listing 5-48.** Adding a Method in the `MyAsyncMethods.cs` File in the Models Folder

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

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;
        }

        public static async Task<IEnumerable<long?>>
            GetPageLengths(List<string> output, params string[] urls) {
            List<long?> results = new List<long?>();
            HttpClient client = new HttpClient();
            foreach (string url in urls) {
                output.Add($"Started request for {url}");
                var httpMessage = await client.GetAsync($"http://{url}");
                results.Add(httpMessage.Content.Headers.ContentLength);
                output.Add($"Completed request for {url}");
            }
            return results;
        }
    }
}
```

The `GetPageLengths` method makes HTTP requests to a series of web sites and gets their length. The requests are performed asynchronously, but there is no way to feed the results back to the method's caller as they arrive. Instead, the method waits until all the requests are complete and then returns all of the results in one go. In addition to the URLs that will be requested, this method accepts a `List<string>` to which I add messages in order to highlight how the code works. Listing 5-49 updates the Index action method of the Home controller to use the new method.

**Listing 5-49.** Using the New Method 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() {
            List<string> output = new List<string>();
            foreach(long? len in await MyAsyncMethods.GetPageLengths(output,
                "apress.com", "microsoft.com", "amazon.com")) {
                output.Add($"Page length: { len}");
            }
            return View(output);
        }
    }
}

```

The action method enumerates the sequence produced by the `GetPageLengths` method and adds each result to the `List<string>` object, which produces an ordered sequence of messages showing the interaction between the `foreach` loop in the `Index` method that processes the results and the `foreach` loop in the `GetPageLengths` method that generates them. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser (which may take several seconds to appear and may have different page lengths):

---

```

Started request for apress.com
Completed request for apress.com
Started request for microsoft.com
Completed request for microsoft.com
Started request for amazon.com
Completed request for amazon.com
Page length: 101868
Page length: 159158
Page length: 91879

```

---

You can see that the `Index` action method doesn't receive the results until all the HTTP requests have been completed. This is the problem that the asynchronous enumerable feature solves, as shown in Listing 5-50.

**Listing 5-50.** Using an Asynchronous Enumerable in the MyAsyncMethods.cs File in the Models Folder

```

using System.Net.Http;
using System.Threading.Tasks;
using System.Collections.Generic;

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;
        }
    }
}

```

```

    public static async IEnumerable<long?>
        GetPageLengths(List<string> output, params string[] urls) {
        HttpClient client = new HttpClient();
        foreach (string url in urls) {
            output.Add($"Started request for {url}");
            var httpMessage = await client.GetAsync($"http://{url}");
            output.Add($"Completed request for {url}");
            yield return httpMessage.Content.Headers.ContentLength;
        }
    }
}

```

The methods result is `IAsyncEnumerable<long?>`, which denotes an asynchronous sequence of nullable long values. This result type has special support in .NET Core and works with standard `yield return` statements, which isn't otherwise possible because the result constraints for asynchronous methods conflict with the `yield` keyword. Listing 5-51 updates the controller to use the revised method.

**Listing 5-51.** Using an Asynchronous Enumerable 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() {
            List<string> output = new List<string>();
            await foreach(long? len in MyAsyncMethods.GetPageLengths(output,
                "apress.com", "microsoft.com", "amazon.com")) {
                output.Add($"Page length: { len}");
            }
            return View(output);
        }
    }
}

```

The difference is that the `await` keyword is applied before the `foreach` keyword and not before the call to the `async` method. Restart ASP.NET Core and request `http://localhost:5000`; once the HTTP requests are complete, you will see that the order of the response messages has changed, like this:

---

```

Started request for apress.com
Completed request for apress.com
Page length: 101868
Started request for microsoft.com
Completed request for microsoft.com
Page length: 159160
Started request for amazon.com
Completed request for amazon.com
Page length: 91674

```

---

The controller receives the next result in the sequence as it is produced. As I explain in Chapter 19, ASP.NET Core has special support for using `IAsyncEnumerable<T>` results in web services, allowing data values to be serialized as the values in the sequence are generated.

## 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 5-52.

**Listing 5-52.** 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;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult 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. Restart ASP.NET Core and request `http://localhost:5000`, and you will see 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
```

---

This approach is prone to errors, either because the name was mistyped or because the code was refactored and the name in the string isn't correctly updated. C# supports the `nameof` expression, in which the compiler takes responsibility for producing a name string, as shown in Listing 5-53.

**Listing 5-53.** 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;
using System.Threading.Tasks;
```

```

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult 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. There is 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 ASP.NET Core 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 that you will see throughout the examples in this book. In the next chapter, I explain how to set up a unit test project for ASP.NET Core.