In-class Lab 04

ASP.NET Core MVC

1 Beginning the lab

1. Create a new project. The target framework should be .NET Core 2.0. Select `File` ▶ `New` ▶ `Project` ▶ `Visual C#` ▶ `Web`. Select `ASP.NET Core Web Application`. Name the application `LanguageFeatures` and save it in your `/aspnetcore/projects` directory. See figure 1. Click `OK`.

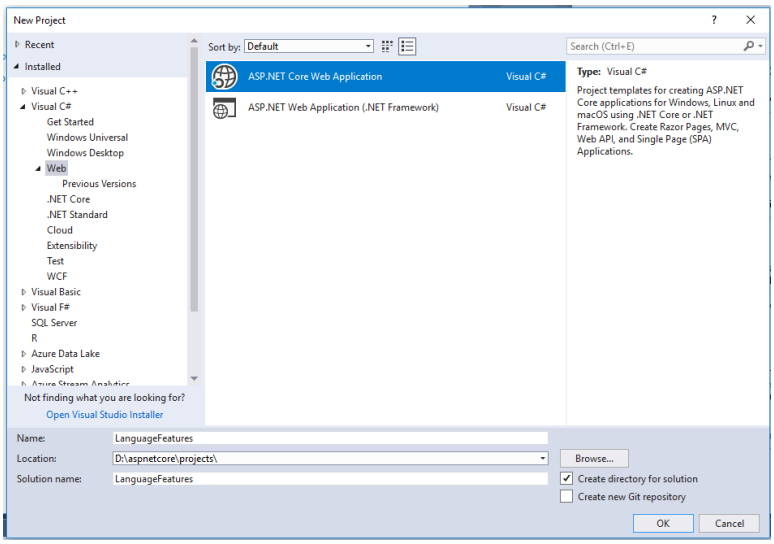

Figure 1: Create a Web Application

2. Select the Empty template. Make sure that **No Authentication** is selected and that Docker support is unselected. See figure 2. Click `OK`.

3. Edit `Startup.cs` like listing 1.

Listing 1: Edit Startup.cs

```
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
```

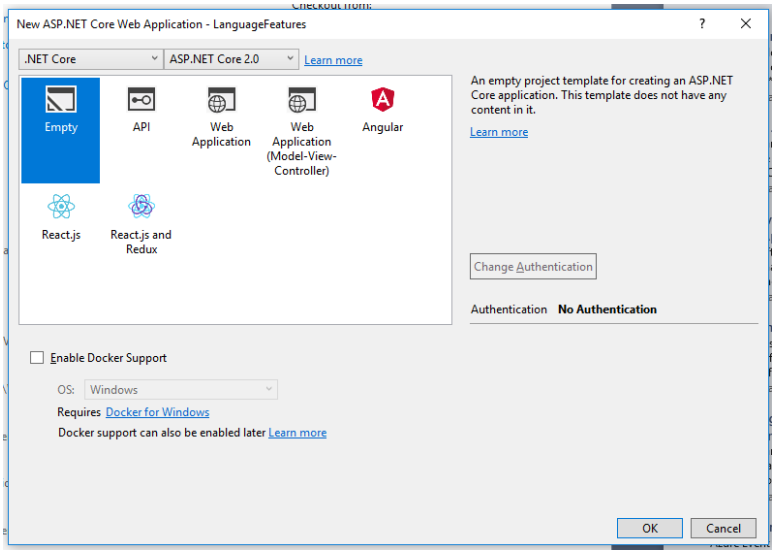

Figure 2: Select the Empty template

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

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

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

4. Create a model by right clicking on the **LanguageFeatures** project and selecting `Add` ▶ `New Folder`. See figure 3. The new folder will appear in the Solution Explorer. Name the new folder `Models`.

5. Create a new Model by right clicking on the `Models` folder and selection `Add` ▶ `Class`. See figure 4. Name the new class `Product.cs`. See figure 5. Click `Add`.

6. Edit the `Product` class like listing 2. Build the project to check for errors.

Listing 2: Edit class Product

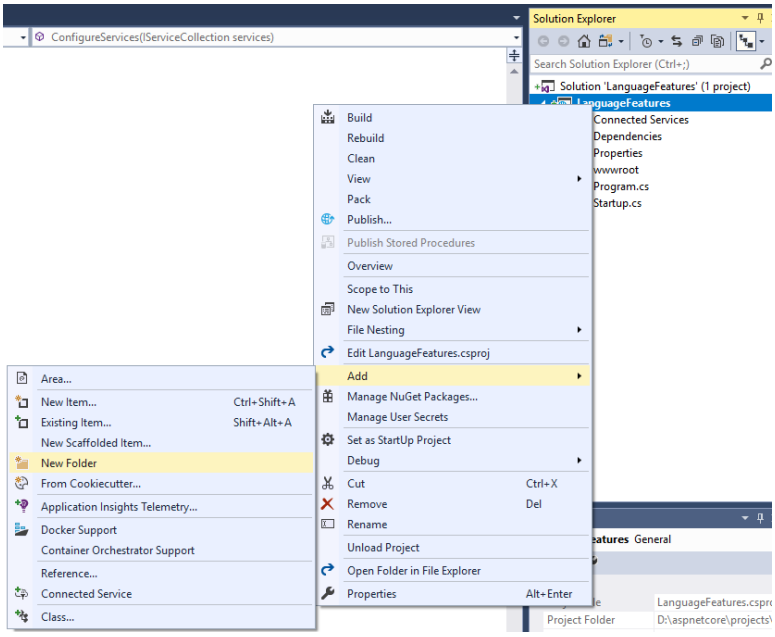

Figure 3: Adding a Models folder

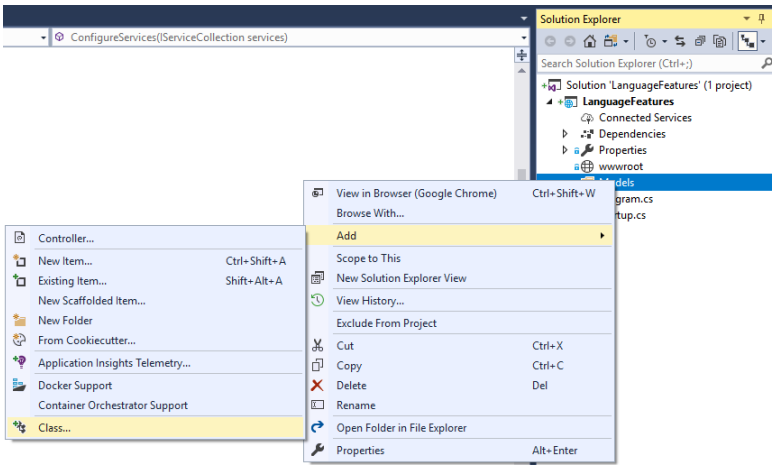

Figure 4: Adding a Model class

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

namespace LanguageFeatures.Models
{
    public class Product
    {
        public string Name { get; set; }
```

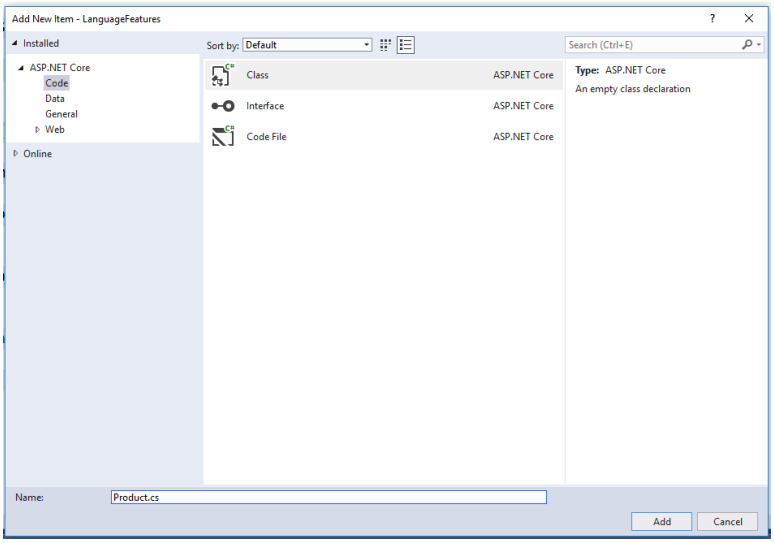

Figure 5: Adding the Model class

```
        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 };
        }
    }
}
```

7. Create a `Controllers` folder under the **LanguageFeatures** project the same way you created the `Models` folder. Make sure you rename the new folder to Controllers.

8. Add a new controller to the `Controllers` folder by right clicking the `Controllers` folder and selecting Add ▶ Controller. See figure 6. Select `MVC Controller - Empty` ▶ `Add`. See figure 7. Name the controller `HomeController` and click `Add`.

9. Edit the `HomeController` like listing 3.

Listing 3: Editing the HomeController

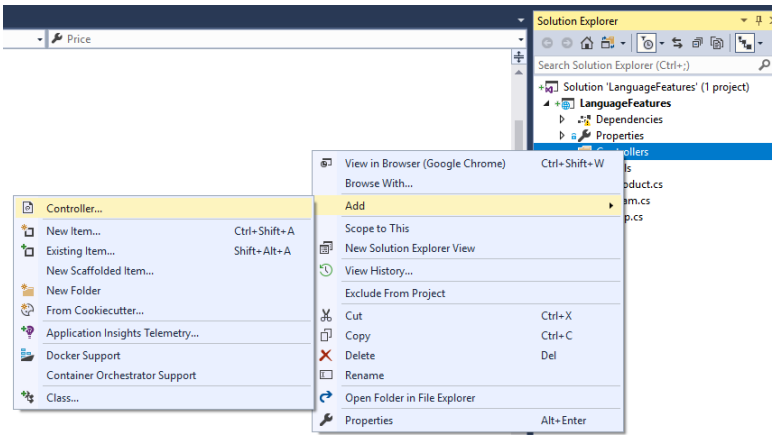

Figure 6: Adding `HomeController`

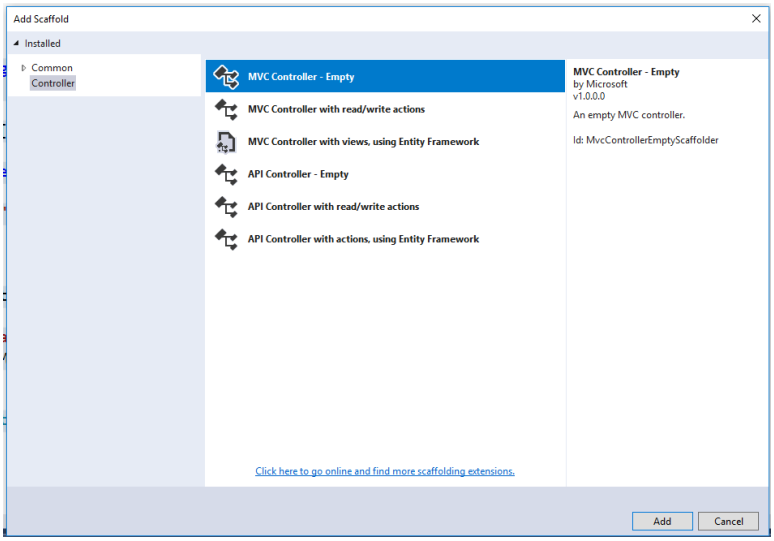

Figure 7: Adding the `HomeController`

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

namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            return View(new string[] {"C#", "Language", "Features"});
        }
```

```
    }
}
```

10. Add a `Views` folder by right clicking the **LanguageFeatures** project and selecting `Add` ▶ `New Folder`. Change the name of the new folder to `Views`. Then, add a sub-folder to `Views`, named `Home`. See figure 8. When you are done, your Solution Explorer should look like figure 9.

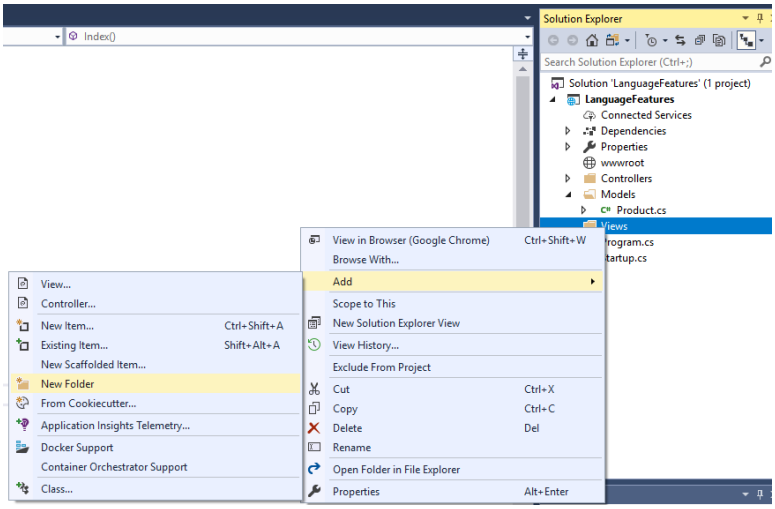

Figure 8: Adding folder `Views/Home`

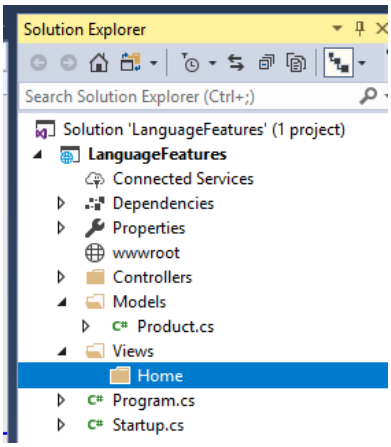

Figure 9: Solution Explorer showing `Views/Home`

11. Add a view named `Index.cshtml` to `Views/Home` by right clicking the `Home` folder and selecting `Add` ▶ `View`. Name the view `Index.cshtml`. See fugure **??**. Edit the view like listing **??**. Start without debgging. What happens? Why? Close the browser window.

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

<!DOCTYPE html>
<html>
```

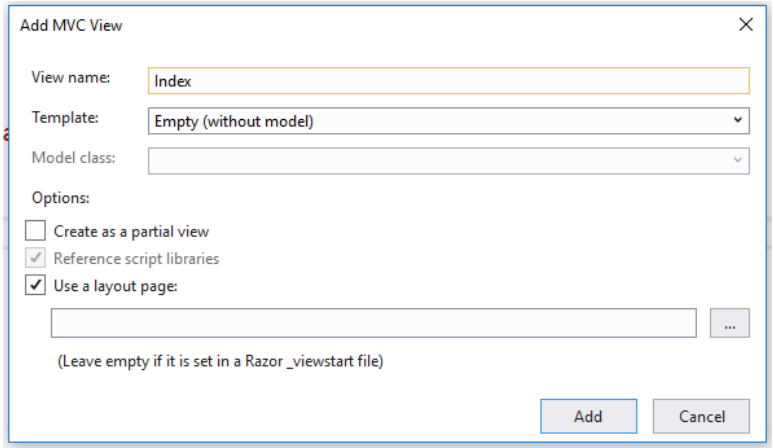

Figure 10: Adding `Index.cshtml`

```
    <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>
```

2 Using the null conditional Operator

12. Edit the `HomeController.cs` file as in listing 4. Start without debugging. What happened? Why?

Listing 4: Edits to `HomeController`

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

namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
```

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

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

13. Edit the `Product.cs` file like listing 5 by addig a new `Related` property and initializing that property for kayak..

Listing 5: Editing `Product.cs`

```
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 };
    }
```

14. In the `HomeControllers.cs` file, edit the `foreach` loop as in listing 6. Start without debugging. What happened? Close the browser window.

Listing 6: Editing the `foreach` loop

```
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 ));
        }
```

15. Finally, in the `HomeControllers.cs` file, edit the `foreach` loop as in listing 7. Start without debugging. What happened? Close the browser window.

Listing 7: Editing the `foreach` loop

```
    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 ));
            }
```

3 Using automatically implemented properties

16. Adding an auto implemented property: Edit the `Product.cs` file to match listing 8

Listing 8: Adding an auto implemented property to `Product.cs`

```
public string Name { get; set; }
*(\cd\bff{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",
        *(\cd\bff{Category = "Water␣Craft",}*)
        Price = 275M
    };
```

17. Adding a read only property: Edit the `Product.cs` file to match listing 9

Listing 9: Adding a read only property to `Product.cs`

```
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;
```

18. Assigning a value to a read only property: Edit the `Product.cs` file to match listing 10

Listing 10: Assigning a value to a read only property to `Product.cs`

```
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 };
    }
}
```

19. In the `HomeControllers.cs` file, edit the `foreach` loop as in listing 11. Start without debugging. What happened? Close the browser window.

Listing 11: Using string interpolation to print variables

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

4 Using object and collection initializers

20. Revise `HomeController.cs` to match listing ??

Listing 12: Revision to `HomeController`, collection initializer

```
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);
}
```

5 Using extension methods

21. Create a new class in the `Models` folder. Name it `ShoppingCart.cs` Edit it to match listing 13. We will extend this class using an extension metnod.

Listing 13: Class `ShoppingCart.cs`

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

namespace LanguageFeatures.Models
{
    public class ShoppingCart
    {
        public IEnumerable<Product> Products { get; set; }
    }
}
```

22. Create a new class in the `Models` folder. Name it `MyExtensionMethods.cs` Edit it to match listing 14. This class extends the `ShoppingCart` class.

Listing 14: The extension class `MyExtensionMethods`

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

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;
        }
    }
```

23. Edit the `HomeControllers.cs` class to match listing 15. Start without debugging. After you examine the results, close the browser window.

Listing 15: Using the extension method

```
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}" });
    }
}
```

24. Now, we apply the extension method to an interface. Edit class `ShoppingCart.cs` as shown in listing 16.

Listing 16: Implementing an interface

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

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();
        }
    }
```

```
}
```

25. Now, edit the `MyExtensionMethods` class as in listing 25.

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

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;
        }
    }
}
```

26. Finally, edit the `HomeController` class as in listing 17.

Listing 17: Edit to class `HomeController`

```
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}"});
    }
}
```

27. Too create a filterig example of an extension method, edit class `MyExtensionMethods` by adding a method `FilterByPrice()` like listing 18.

Listing 18: Filtering extension method, `FilterByPrice()`

```
public static IEnumerable<Product> FilterByPrice( this IEnumerable<Product> productEnum,
    decimal minimumPrice)
{
    foreach (Product prod in productEnum)
    {
        if ((prod?.Price ?? 0) >= minimumPrice)
        {
            yield return prod;
        }
    }
}
```

28. Then, edit class `HomeControllers.cs` as in listiing 19.

Listing 19: Revised class `HomeControllers.cs`

```
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 priceTotal = productArray.FilterByPrice(20).TotalPrices();
    return View("Index", new string[] { $"Array Total:  {priceTotal:C2}" });
}
```

6 Using lambda expressions

29. To add a `FilterByName()` method, add the method in listing 20 to `MyExtensionMethods`.

Listing 20: The `FilterByName()` method

```
public static IEnumerable<Product> FilterByName(this IEnumerable<Product> productEnum, char
    firstLetter)
{
    foreach (Product prod in productEnum)
    {
        if ((prod?.Name?[0]) == firstLetter)
        {
            yield return prod;
        }
    }
}
```

30. Complete the name filtering by revising the `Homeontroller.cs` class as in listing 21.

Listing 21: Revision to HomeController

```
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 priceTotal = productArray.FilterByPrice(20).TotalPrices();
    decimal nameFilter = productArray.FilterByName('S').TotalPrices();

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

31. To use a lambda expression to filter products, and to generalize the filtering function, first add a `Filter()` method to `MyExtensionMethods`, as shown in listing 22.

Listing 22: The `Filter()` method

```
public static IEnumerable<Product> Filter(this IEnumerable<Product> productEnum, Func<
    Product, bool> selector)
{
    foreach (Product prod in productEnum)
    {
        if (selector(prod))
        {
            yield return prod;
        }
    }
}
```

32. Then, make the following changes in class `HoneController` to complete the generalization of the filter function with lambda expressions, shown in listing 23.

Listing 23: Lambda expression in `HomeController`

```
 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[]  $"Array Total:  priceFilterTotal:C2", $"Name Total:
    nameFilterTotal:C2" );
}
```

7 Using anonymous types

33. To illustrate anonymous types, the use of the `var` ketword, edit `HoneController.cs` as shown in listing 24. What is the result when you run this code?

Listing 24: Use of the keyword `var`

```
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));
}
```

34. Wyhen you use `var`, what is the type of the object? In order to see the type, change the `return` statement to **return View(products.Select(p => p.GetType().Name));** Run the code, and explain the result.

8 Gettiing names

35. Chate the `return` statement in `HomeController.cs` to matching listing 25. What happens when you run this? Why?

Listing 25: Use of `nameof()` method

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