C# FEATURES

SUCCINCTLY

BY DIRK STRAUSS

SUCCINCTLY EBOOK SERIES

Syncfusion®

C# Features Succinctly

By

Dirk Strauss

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-209-6

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Dirk Strauss is a software developer from South Africa with over 13 years of programming experience. He has extensive experience in SYSPRO customization, with a focus on C# and web development. He studied at Nelson Mandela University, where he wrote software part-time to gain a better understanding of the technology. He remains passionate about writing code and sharing what he learns with others.

Chapter 1 The History of C#

C# is continuing to evolve and improve with each major release of the language. As the C# team is innovating and adding features to C#, its members share their thought process around the design of C# with the community.

If you head over to the dotnet/Roslyn repository on GitHub, you will see detailed language feature statuses as well as features that the C# team is considering for upcoming releases. You can also view a history C# on the Microsoft Docs. As of this writing, of particular interest are the planned features surrounding C# 9. Later in this book, we will have a look at what is planned for C# 9.

The C# build tools will default the language version to the latest major release. As of C# 7.0, however, developers started seeing more point releases in the form of versions C# 7.1, C# 7.2, and C# 7.3.

C# language version

When creating a new project in Visual Studio, the C# compiler figures out which version of C# to use based on the target framework of your project. This ensures you do not use a language version that requires types or runtime behavior not available in the target framework of the project.

It is worth mentioning that C# 8.0 and higher will only be supported on .NET Core 3.x and later. Table 1 outlines the C# language defaults based on the target framework.

Table 1: Language version default mapping

Target framework	Version	C# language default
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	All	C# 7.3

As you will see later in this book, you can override the default language version. You can do this by:

- Manually editing your .csproj file (explained later).

- Configuring multiple projects using a Directory.Build.props file.
- Configuring the **`-langversion`** compiler option.

For more on the **`-langversion`** compiler option, refer to this .NET documentation.

C# version 1.0

The release of Visual Studio .NET 2002 brought with it C# 1.0, a Java-looking, general-purpose, object-oriented language. It didn't contain LINQ or generics, but what it did offer developers was a viable alternative to Java, on a Windows platform.

C# 1.0 included the following:

- Classes
- Structs
- Interfaces
- Events
- Properties
- Delegates
- Expressions
- Statements
- Attributes

Having a look at these features today, one would be excused for feeling a little spoiled with the features available in C# 8.0.

C# version 1.2

Visual Studio .NET 2003 shipped with C# 1.2 and only contained a few small feature enhancements. It is here that code generated in **`foreach`** loops would call **`Dispose`** on an **`IEnumerator`** if it implemented **`IDisposable`**.

C# version 2.0

When Visual Studio 2005 was released in 2005, we saw some nice features, such as:

- Generics
- Partial classes
- Anonymous methods
- Nullable value types
- Iterators
- Covariance and contravariance

Existing features were also improved, such as:

- Getter/setter separate accessibility.

- Method group conversions (simply assigning the name of a method to a delegate without using a new operator).
- Static classes.
- Delegate inference.

In many respects, C# 2.0 was a turning point in the language, especially with the introduction of generics and iterators.

C# version 3.0

In the latter part of 2007, Visual Studio 2008 was released. The language features it provided came with the release of the .NET Framework 3.5. These were as follows:

- Auto-implemented properties.
- Anonymous types.
- Query expressions (LINQ).
- Lambda expressions.
- Expression trees.
- Extension methods.
- Implicitly typed local variables.
- Partial methods.
- Object and collection initializers.

LINQ was one of the killer features of C#. This allowed you to write SQL-style, declarative queries on collections, for example. Initially, I struggled with LINQ (and I'm not the only one). These days, it is just quicker and more concise to write. C# 3.0 was truly a groundbreaking release.

C# version 4.0

Visual Studio 2010 brought C# 4.0. The features added in version 4.0 were:

- Dynamic binding.
- Named/optional arguments.
- Generic covariant and contravariant.
- Embedded interop types.

For me, the introduction of optional parameters was a breath of fresh air, but it was the addition of the **`dynamic`** keyword in version 4.0 that stole the show.

C# version 5.0

C# version 5.0 was released with Visual Studio 2012 and made another groundbreaking addition to the C# language—the introduction of **`async`** and **`await`** for asynchronous programming. C# 5.0 introduced:

- Asynchronous members

- Caller info attributes

Asynchrony was now baked into C# as a first-class citizen.

C# version 6.0

Visual Studio 2015 brought C# version 6.0, and it is here that we started seeing smaller features that made developers more productive. These included:

- using static (no more Console.this and Console.that).
- Read-only auto-properties.
- Auto-property initializers.
- Expression-bodied function members.
- Null-conditional operators.
- String interpolation.
- Exception filters.
- The **`nameof`** expression.
- **`await`** in **`Catch`** and **`Finally`** blocks.

I remember Mads Torgersen saying that C# 6.0 added a lot of syntactic sugar to the language. I agree with him 100 percent.

C# version 7.0

Visual Studio 2017 saw C# version 7.0 released to the developer community. It introduced the following language features:

- **`out`** variables.
- Tuples.
- Discards.
- Pattern matching.
- **`ref`** locals and returns.
- Local functions.
- More expression-bodied members.
- **`throw`** expressions.
- Generalized async return types.
- Numeric literal syntax improvements.

Developers could now write cleaner code and be more productive.

C# version 7.1

With C# 7.0, we started seeing point releases on C#, starting with version 7.1. This marked an increased release cadence for C#. New language features for this release were:

- **`async Main`** method.
- default literal expressions.

- Inferred tuple element names.
- Pattern matching on generic type parameters.

C# 7.1 also added the language version selection configuration element, as well as new compiler behavior.

C# version 7.2

C# 7.2 added a few more smaller language features to C#. These were:

- The addition of code enhancements allowing developers to write safe, efficient code:
  - The **`in`** modifier on parameters.
  - The **`ref readonly`** modifier on method returns.
  - The **`readonly struct`** declaration.
  - The **`ref struct`** declaration.
- Non-trailing named arguments.
- Leading underscores in numeric literals.
- **`private protected`** access modifier.
- Conditional **`ref`** expressions.

You can read more on writing safe and efficient code in C# here.

C# version 7.3

The point releases of C# 7 allowed developers to get their hands onto new language features sooner rather than later. It was the release of C# 7.3 that had two main themes. One theme allowed safe code to be as performant as unsafe code, and the other provided additional improvements to existing features.

From a better performant safe code perspective, we saw:

- The accessing of fields without pinning.
- Reassigning **`ref`** local variables.
- The use of initializers on **`stackalloc`** arrays.
- Using **`fixed`** statements with any type that supports a pattern.
- Additional generic constraints (you will see this in action with the unmanaged constraint later on in this book).

From an enhancement perspective, we saw:

- Being able to compare tuple types with **`==`** and **`!=`**.
- The use of expression variables in more locations.
- Attaching attributes to the backing field of auto-implemented properties.
- Improved method resolution when arguments differ by **`in`**.
- Fewer ambiguous cases for overload resolution.

We also saw new compiler options: **`-publicsign`** to enable open-source software signing of assemblies, and **`-pathmap`** to provide a mapping for source directories.

C# version 8.0

C# 8.0 specifically targets .NET Core. This is what this book is all about, and why I would imagine you are reading it.

The following features and enhancements were added to C# 8.0:

- Readonly members.
- Default interface methods.
- Pattern matching enhancements.
- Using declarations.
- Static local functions.
- Disposable ref structs.
- Nullable reference types.
- Asynchronous streams.
- Indices and ranges.
- Null-coalescing assignment.
- Unmanaged constructed types.
- Stackalloc in nested types.
- Enhancements to interpolated verbatim strings.

There is a lot to consume here. If you have little experience with C#, the best place to start is C# 7. This book will take a look at C# 7 before delving into the new features of C# 8.0.

If you are already comfortable with C# 7, then dive straight into the chapter on C# 8.0. Are you ready? Let's go.

Chapter 2 C# 7 Features Recap

To appreciate what C# 8.0 has to offer developers, it is important that we revisit some of the features available in C# 7. The C# language has been evolving, but it feels like the incremental releases of C#, which bring new features to the language, have been increasing in recent years.

> ***Note: Most of the code examples in this ebook are presented as instance methods rather than static methods. If you wish to experiment with the example methods, you may want to decorate the examples with the `static` modifier so you can call them directly as opposed to placing them inside a class definition, instantiating an object instance, and then calling the method.***

Please note that the **`public Demo`** used in some of the code samples is the constructor for a class called **`Demo`**. I encourage you to download the code for this book from GitHub.

out variables

C# allows developers to use **`out`** variables. You might have used them before, and they are quite handy in certain circumstances. One thing that has always bugged me, though (before C# 7 was released), was the need for creating a "loose-hanging" variable. Consider Code Listing 1. The declaration of the **`numberOfCopies`** variable in the constructor of the **`Demo`** class was necessary in order to use **`out`** variables in C#.

Code Listing 1: The out variable before C# 7

```
public Demo()
{
    int numberOfCopies = 0;
    GetNumberOfCopies(out numberOfCopies);
}

private void GetNumberOfCopies(out int numCopies)
{
    numCopies = 20;
}
```

Here is this (in my opinion) ugly-looking code required to make use of a truly neat language feature in C#. Along comes C# 7, and shakes things up a little with an improved syntax for **`out`** variables. Let's consider the same code (in the constructor of the **`Demo`** class) from Code Listing 1, and rewrite it slightly:

Code Listing 2: The out variable in C# 7

```
public Demo()
{
    GetNumberOfCopies(out int numberOfCopies);
}

private void GetNumberOfCopies(out int numCopies)
{
    numCopies = 20;
}
```

You will notice that C# 7 allowed developers to get rid of the unnecessary variable declaration **`int numberOfCopies = 0;`** and declare the **`out`** variable in the argument list of the **`GetNumberOfCopies`** method call.

This results in much cleaner code and makes the code easier to read. But C# 7 allows developers to go one step further. If you have been following along in Visual Studio, you will see a green squiggly line under the **`int`** in your argument list of the **`GetNumberOfCopies`** method call.

Note: If you do not see a code style suggestion, you might have suppressed this code style rule in an EditorConfig file or your code style preferences. For more information on code style preferences, see this article in Microsoft Docs.

This squiggly line is suggesting that you replace the **`int`** with the **`var`** keyword, as seen in Code Listing 3.

Code Listing 3: Replacing the int with var

```
public Demo()
{
    GetNumberOfCopies(out var numberOfCopies);
}

private void GetNumberOfCopies(out int numCopies)
{
    numCopies = 20;
}
```

If you think about it, this is perfectly logical. The type of **`numberOfCopies`** is inferred from the type in the method signature of the **`GetNumberOfCopies`** method. This syntax can be used for any **`out`** variable, such as in a **`TryParse`**.

Code Listing 4: The out variable in a TryParse

```
if (int.TryParse("3", out var factor))
{
```

```
}
```

Code Listing 4 illustrates the use of the **out** variable in a **TryParse**. The benefit of using the improved syntax for **out** variables is:

- It improves code readability by declaring the **out** variable only where you use it.
- Because the declaration happens where you use it, you need not assign an initial value.

The syntax change to the **out** variable in C# 7 is but one of the improved language features aimed at making your code more readable and concise.

Discards

In the previous section, we had a look at **out** variables. We saw that we can declare the **out** variable right where we need it, without having to assign an initial value. What if we don't care about the value assigned to the **out** variable?

C# now supports discards to allow developers to indicate that they don't care for the variable. The discard is a write-only variable, and is denoted by using an underscore **_** in your assignment.

Think of the discard as an unassigned variable. It can be used in the following situations:

- When used as **out** parameters.
- With the **is** and **switch** statements.
- As a standalone identifier.
- During the deconstruction of tuples (more on tuples in the next section) or user-defined types.

Let's have a look back at Code Listing 4. If all we want to do is check if the value parses to an integer value, we can use a discard variable instead of declaring the variable **factor**, as shown in Code Listing 5.

Code Listing 5: Using a discard

```
if (int.TryParse("3", out var _))
{

}
```

I like using extension methods. A favorite use for extension methods is to check if a string value is an integer. Admittedly, the extension method can do so much more than the simple example in Code Listing 6. What I want to focus your attention on, however, is the use of the discard variable here.

Code Listing 6: Using a discard in an extension method

```
public static bool ToInt(this string value)
{
    return int.TryParse(value, out var _);
}
```

I don't care about the value—I just want to check if it's an integer—and the discard is a perfect candidate for this type of situation.

Tip: You can further simplify the code in Code Listing 6 by using an expression body.

Tuples

Sometimes developers need to pass structures containing multiple data elements to methods. Tuples were added to C# to provide data structures containing multiple fields (up to a maximum of eight items) representing the data members.

C# 7.0 also introduced language support for tuples that enabled semantic names for the tuple fields using new tuple types. Code Listing 7 illustrates a basic example of a tuple.

Code Listing 7: Field names specified in tuple initialization expression

```
var foundedDates = (Microsoft: 1975, Apple: 1976, Amazon: 1994);

Console.WriteLine($"Microsoft founded in {foundedDates.Microsoft}");
Console.WriteLine($"Apple founded in {foundedDates.Apple}");
Console.WriteLine($"Amazon founded in {foundedDates.Amazon}");
```

Here in Code Listing 7, you can see that the field names are explicitly specified in the tuple initialization expression. You can also specify the field names in the tuple type definition, as seen in Code Listing 8.

Code Listing 8: Field names specified in the type definition

```
(int Microsoft, int Apple, int Amazon) foundedDates = (1975, 1976, 1994);

Console.WriteLine($"Microsoft founded in {foundedDates.Microsoft}");
Console.WriteLine($"Apple founded in {foundedDates.Apple}");
Console.WriteLine($"Amazon founded in {foundedDates.Amazon}");
```

C# will also allow you to infer the field names from the variable names in the tuple initialization expression. This is illustrated in Code Listing 9.

Code Listing 9: Tuple field names inferred

```
var distanceToEarth = 384400;
var radius = 1737.1;
var moon = (distanceToEarth, radius);

Console.WriteLine($"The moon is {moon.distanceToEarth} km from Earth.");
Console.WriteLine($"The moon has a radius of {moon.radius} km.");
```

The field names are therefore inferred, so it's probably a good idea think about them before arbitrarily naming variables. For example, if **`distanceToEarth`** was simply named **`distance`**, then the tuple field would read as **`moon.distance`**, which somewhat obfuscates the intent.

Tuple equality

Tuple types also have support for **`==`** and **`!=`** operators. This means that the code in Code Listing 10 will equate to **`true`**.

Code Listing 10: Comparing tuples

```
var teamOne = (JohnScore: 15, MikeScore: 27);
var teamTwo = (SallyScore: 15, MelissaScore: 27);

Console.WriteLine(teamOne == teamTwo); // Equates to true
```

You can only compare tuples when:

- Each tuple has the same number of elements. If **`teamOne`** had an additional score, then Visual Studio would tell you that the tuple types must have matching cardinalities.
- For every tuple position, the elements from the left-hand and right-hand tuple operands are comparable with the **`==`** and **`!=`** operators.

This means the following code would not be comparable.

Code Listing 11: Non-comparable tuples

```
var teamOne = (JohnScore: 15, MikeScore: "27");
var teamTwo = (SallyScore: 15, MelissaScore: 27);

Console.WriteLine(teamOne == teamTwo); // Results in a compile-time error
```

This is because **`==`** cannot be applied to operands of type **`string`** and **`int`**.

Using a tuple as a method return type

Methods can also return tuple types. Consider the method illustrated in Code Listing 12.

Code Listing 12: Method returning a tuple

```
private (int Age, DateTime BirthDate, string Fullname) ReadPersonInfo()
{
    var personData = (age: 0, birthday: DateTime.MinValue, fullName: "");
    // Read data from somewhere
    personData.fullName = "Joe Soap";

    var today = DateTime.Now;
    personData.birthday = today.AddYears(-44);
    personData.age = today.Year - personData.birthday.Year;

    return personData;
}
```

The method simply returns a tuple, which can then be used by the calling code, as illustrated in Code Listing 13.

Code Listing 13: Calling method with tuple return type

```
var person = ReadPersonInfo();
Console.WriteLine($"{person.Fullname} was born on {person.BirthDate:dd MMMM
yyyy} and is {person.Age} years old.");
```

If you wanted to, you could also deconstruct the tuple instance into separate variables, as illustrated in Code Listing 14.

Code Listing 14: Deconstruct tuples into explicit variable types

```
(int age, DateTime DOB, string fullName) = ReadPersonInfo();
Console.WriteLine($"{fullName} was born on {DOB:dd MMMM yyyy} and is {age}
years old.");
```

This allows me to explicitly specify the variable types to deconstruct the tuple into. I can also let the compiler do all the work for me by implicitly declaring the deconstructed variables. For this, I can use the **var** keyword, as illustrated in Code Listing 15.

Code Listing 15: Using the var keyword for implicit deconstruction

```
var (age, DOB, fullName) = ReadPersonInfo();
Console.WriteLine($"{fullName} was born on {DOB:dd MMMM yyyy} and is {age}
years old.");
```

This is great for when you are not sure of the return type, or if you simply don't want to specify the return type for each deconstructed variable.

Pattern matching

Patterns in C# test whether a value has a certain shape. When one hears the word “test,” one thinks of **`if`** or **`switch`** statements in C#. When the test results in a match, that value being tested can be used to extract information.

Consider the following classes.

Code Listing 16: Shape Classes

```
public class Cylinder
{
    public double Length { get; }
    public double Radius { get; }

    public Cylinder(double length, double radius)
    {
        Length = length;
        Radius = radius;
    }
}

public class Sphere
{
    public double Radius { get; }

    public Sphere(double radius)
    {
        Radius = radius;
    }
}

public class Pyramid
{
    public double BaseLength { get; }
    public double BaseWidth { get; }
    public double Height { get; }

    public Pyramid(double baseLength, double baseWidth, double height)
    {
        BaseLength = baseLength;
        BaseWidth = baseWidth;
        Height = height;
    }
}
```

Using the **`is`** type pattern expression, we can check what the type of the **`volumeShape`** variable is, and then perform a specific action based on that type to calculate the volume. This is illustrated in Code Listing 17 in a generic method called **`CalculateVolume`**.

Code Listing 17: Using is type pattern expression

```
private double CalculateVolume<T>(T volumeShape)
{
    if (volumeShape is Cylinder c)
        return Math.PI * Math.Pow(c.Radius, 2) * c.Length;
    else if (volumeShape is Sphere s)
        return 4 * Math.PI * Math.Pow(s.Radius, 3) / 3;
    else if (volumeShape is Pyramid p)
        return p.BaseLength * p.BaseWidth * p.Height / 3;

    throw new ArgumentException(message: "Unrecognized object", paramName:
nameof(volumeShape));
}
```

Calling the generic **`CalculateVolume`** method is done as illustrated in Code Listing 18.

Code Listing 18: Calling the CalculateVolume method

```
var cylinder = new Cylinder(20, 2.5);
var sphere = new Sphere(2.5);
var pyramid = new Pyramid(2.5, 3, 16);

var cylinderVol = CalculateVolume(cylinder);
var sphereVol = CalculateVolume(sphere);
var pyramidVol = CalculateVolume(pyramid);

Console.WriteLine($"The volume of the Cylinder is
{Math.Round(cylinderVol,2)}");
Console.WriteLine($"The volume of the Sphere is {Math.Round(sphereVol,
2)}");
Console.WriteLine($"The volume of the Pyramid is
{Math.Round(pyramidVol,2)}");
```

This allows developers to simplify their code and make it more readable. We can also apply pattern matching to **`switch`** statements, as illustrated in Code Listing 19.

Code Listing 19: Using pattern matching switch statements

```
private double CalculateVolume<T>(T volumeShape)
{
    switch (volumeShape)
    {
        case Cylinder c:
            return Math.PI * Math.Pow(c.Radius, 2) * c.Length;
        case Sphere s:
            return 4 * Math.PI * Math.Pow(s.Radius, 3) / 3;
        case Pyramid p:
```

```
            return p.BaseLength * p.BaseWidth * p.Height / 3;
        default:
            throw new ArgumentException(message: "Unrecognized object",
paramName: nameof(volumeShape));
    }
}
```

Traditionally, the **switch** statement supported the constant pattern, allowing you to compare a variable to any constant in the **case** statement. This was limited to numeric and **string** types. In C# 7 those restrictions don't apply anymore, and you can use type patterns in **switch** statements.

Furthermore, you can also use **when** clauses in your **case** expressions. Consider the code in Code Listing 20.

Code Listing 20: Using when clauses in case expressions

```
private double CalculateVolume<T>(T volumeShape)
{
    switch (volumeShape)
    {
        case Sphere s when s.Radius == 0:
            return 0;
        case Cylinder c:
            return Math.PI * Math.Pow(c.Radius, 2) * c.Length;
        case Sphere s:
            return 4 * Math.PI * Math.Pow(s.Radius, 3) / 3;
        case Pyramid p:
            return p.BaseLength * p.BaseWidth * p.Height / 3;
         default:
            throw new ArgumentException(message: "Unrecognized object",
paramName: nameof(volumeShape));
    }
}
```

The change is subtle, but syntactically important. The **case** statement for the **Sphere** reads as **case Sphere s when s.Radius == 0:** which tells the compiler something important regarding the variable **s**. If the **Radius** of the variable **s** is equal to **0**, do not even attempt the volume calculation, because it makes no difference. The result will always be **0**, so just return **0**.

Local functions

Local functions are by far one of my favorite features of C# 7. When the use of a specific method makes sense in only a single place, then it can be easily made local to only that specific enclosing method.

So if a method called **CalculateVolume** is only used by a single method called **TotalObjectVolume**, then it can be made local to **TotalObjectVolume**. Let's illustrate this by using some simplified code.

Code Listing 21: Simplified code

```
private string MethodOne()
{
    var getText = MethodTwo();
    return getText;
}

private string MethodTwo()
{
    return "I am method two";
}
```

As you would expect, calling **MethodOne** with **Console.WriteLine(MethodOne());** results in the text **I am method two** being displayed.

Introducing a local function called **MethodTwo** inside the body of **MethodOne** will now result in the text **I am local function two** being displayed.

Code Listing 22: Introducing a local function

```
private string MethodOne()
{
    string MethodTwo()
    {
        return "I am local function two";
    }

    var getText = MethodTwo();
    return getText;
}

private string MethodTwo()
{
    return "I am method two";
}
```

This means local functions take precedence when used by code inside the scope of the enclosing method. If the use of **MethodTwo** only made sense from within **MethodOne**, then it would do fine as a local function.

Let's swing back to the **TotalObjectVolume** method that uses the **CalculateVolume** local function, illustrated in Code Listing 23.

Code Listing 23: CalculateVolume local function

```
private double TotalObjectVolume((Cylinder c, Sphere s, Pyramid p)
volumeShapes)
{
    var cylinderVol = CalculateVolume(volumeShapes.c);

    double CalculateVolume<T>(T volumeShape)
    {
        switch (volumeShape)
        {
            case Sphere s when s.Radius == 0:
                return 0;
            case Cylinder c:
                return Math.PI * Math.Pow(c.Radius, 2) * c.Length;
            case Sphere s:
                return 4 * Math.PI * Math.Pow(s.Radius, 3) / 3;
            case Pyramid p:
                return p.BaseLength * p.BaseWidth * p.Height / 3;
            default:
                throw new ArgumentException(message: "Unrecognized object",
paramName: nameof(volumeShape));
        }
    }

    var sphereVol = CalculateVolume(volumeShapes.s);
    var pyramidVol = CalculateVolume(volumeShapes.p);

    return Math.Round(cylinderVol + sphereVol + pyramidVol, 2);
}
```

The **TotalObjectVolume** method takes a tuple called **volumeShapes** as a parameter and uses the local function **CalculateVolume** to calculate the volume of the **Cylinder**, **Sphere**, and **Pyramid** types.

It also allows us to call the local function anywhere inside the enclosing **TotalObjectVolume** method. You can even place the local function after the **return** statement, as seen in Code Listing 24.

Code Listing 24: Local function after return

```
private double TotalObjectVolume((Cylinder c, Sphere s, Pyramid p)
volumeShapes)
{
    var cylinderVol = CalculateVolume(volumeShapes.c);
    var sphereVol = CalculateVolume(volumeShapes.s);
    var pyramidVol = CalculateVolume(volumeShapes.p);

    return Math.Round(cylinderVol + sphereVol + pyramidVol, 2);
```

```
    // Local functions here
    double CalculateVolume<T>(T volumeShape)
    {
        switch (volumeShape)
        {
            case Sphere s when s.Radius == 0:
                return 0;
            case Cylinder c:
                return Math.PI * Math.Pow(c.Radius, 2) * c.Length;
            case Sphere s:
                return 4 * Math.PI * Math.Pow(s.Radius, 3) / 3;
            case Pyramid p:
                return p.BaseLength * p.BaseWidth * p.Height / 3;
            default:
                throw new ArgumentException(message: "Unrecognized object",
paramName: nameof(volumeShape));
        }
    }
}
```

Local functions are a fantastic addition to C# that allow developers to be quite specific in their intent. If you see a local function, then you know that it only makes sense for use within the enclosing method.

Expression-bodied members for constructors and finalizers

C# 6 introduced developers to expression-bodied members. These only apply to member functions and read-only properties. With C# 7 we can now use expression-bodied members on constructors and destructors, as well as on **`get`** and **`set`** accessors on properties and indexers. Consider the **`Circle`** class illustrated in Code Listing 25.

The class contains a constructor and a destructor, as well as a property that returns the square of the radius.

Code Listing 25: The Circle class

```
public class Circle
{
    public double Radius { get; }
    public double RadiusSquared
    {
        get
        {
            return Math.Pow(Radius, 2);
        }
```

```
    }

    public Circle(double radius)
    {
        Radius = radius;
    }

    ~Circle()
    {
        Console.WriteLine("Run cleanup statements");
    }
}
```

Using expression-bodied members, we can cut down on unnecessary code and make the class very readable. Consider the modified **`Circle`** class in Code Listing 26.

Code Listing 26: The Circle class using expression-bodied members

```
public class Circle
{
    public double Radius { get; }
    public double RadiusSquared
    {
        get => Math.Pow(Radius, 2);
    }

    public Circle(double radius) => Radius = radius;

    ~Circle() => Console.WriteLine("Run cleanup statements");
}
```

The code is more readable and succinct.

Generalized async return types

Let's briefly discuss **`async`** methods before C# 7. Every **`async`** method was required to return **`Task`**, **`Task<T>`**, or **`void`**. The use of void-returning methods should only be used with **`async`** event handlers. Generally, an event handler is a case of fire and forget: I don't care what the result of the event is.

If an async method is not returning a value, then **`Task`** is used. If the method does return a value, then **`Task<T>`** is used.

Since **`Task`** is a reference type, an object is allocated when using it. In situations where the **`async`** method will return a cached result or complete synchronously, these additional allocations can impact performance.

In C# 7, the **ValueTask** type has been added to solve this problem.

Note: To make use of `ValueTask`, you must install the System.Threading.Tasks.Extensions NuGet package.

Your **async** methods return types are no longer limited to **Task**, **Task<T>**, and **void**. Have a look at Code Listing 27 and Code Listing 28, which illustrate this language feature.

In Code Listing 27, we have a **static** class that will act as the cache.

Code Listing 27: The static cache class

```
public static class ValueCache
{
    public static int CachedValue { get; set; } = 0;
    public static DateTime TimeToLive { get; set; } = DateTime.MinValue;
}
```

I have added **Console.WriteLine** statements throughout the code to make the output clearer. The code only calls the **DoSomethingAsync** method when the **TimeToLive** value has expired. If the **TimeToLive** is still valid, the cached result is returned.

Code Listing 28: Using ValueTask

```
static async Task Main()
{

    Console.WriteLine(await GetSomeValueAsync());
    Console.WriteLine($"Wait 1 second");
    await Task.Delay(1000);
    Console.WriteLine("");

    Console.WriteLine(await GetSomeValueAsync());
    Console.WriteLine($"Wait 7 seconds");
    await Task.Delay(7000);
    Console.WriteLine("");

    Console.WriteLine(await GetSomeValueAsync());

    _ = Console.ReadLine();
}

public static async ValueTask<int> GetSomeValueAsync()
{
    Console.WriteLine($"DateTime.Now = {DateTime.Now.TimeOfDay}");
    Console.WriteLine($"ValueCache.TimeToLive =
{ValueCache.TimeToLive.TimeOfDay}");
```

```
    if (DateTime.Now <= ValueCache.TimeToLive)
    {
        Console.WriteLine($"Return Cached value");
        return ValueCache.CachedValue;
    }

    var val = await DoSomethingAsync();
    ValueCache.CachedValue = val;
    Console.WriteLine($"Set time to live at 5 seconds");
    ValueCache.TimeToLive = DateTime.Now.AddSeconds(5.0);

    Console.WriteLine($"Return value");
    return val;
}

private static async Task<int> DoSomethingAsync()
{
    await Task.Delay(1);
    return DateTime.Now.Second;
}
```

You can see the output of this in Code Listing 29.

Code Listing 29: Console output

```
DateTime.Now = 17:16:39.8923332
ValueCache.TimeToLive = 00:00:00
Set time to live at 5 seconds
Return value
39
Wait 1 second

DateTime.Now = 17:16:40.9243327
ValueCache.TimeToLive = 17:16:44.9223377
Return Cached value
39
Wait 7 seconds

DateTime.Now = 17:16:47.9273331
ValueCache.TimeToLive = 17:16:44.9223377
Set time to live at 5 seconds
Return value
47
```

It must be noted that the returned type still needs to satisfy the **`async`** pattern. This means that the **`GetAwaiter`** method must be accessible.

Chapter 3 C# 8.0 Features

With the release of C# 8.0, developers have been given more features and enhancements to improve their codebases with, such as pattern matching enhancements. This will become evident when we look at switch expressions later on in this book.

C# 8.0 is supported on .NET Core 3.x and .NET Standard 2.1.

Default interface methods

This change to interfaces in C# 8.0 might be somewhat controversial for some, depending on your views. The logic, however, behind the feature in C# 8.0 is welcome. To understand the change, we need to explain a scenario.

An application creates orders. For this scenario, an interface called **`IOrder`** has been created, and is implemented by your application. This interface is also used in an external codebase maintained by a different team of developers. The interface and implementation look as illustrated in Code Listing 30.

Code Listing 30: The IOrder Interface

```
public class SalesOrder : IOrder
{
    public void CreateOrder(DateTime orderDate) { }
}

public interface IOrder
{
    void CreateOrder(DateTime orderDate);
}
```

Changes to some logic in the program require developers to be able to default the order date. The change, therefore, needs to be made in the interface to provide the ability to create an order without specifying a date. This will then simply default to the current date.

The problem with this approach is that once interfaces are released, they are considered immutable. Adding logic to the **`IOrder`** interface is a breaking change, as seen in Figure 1.

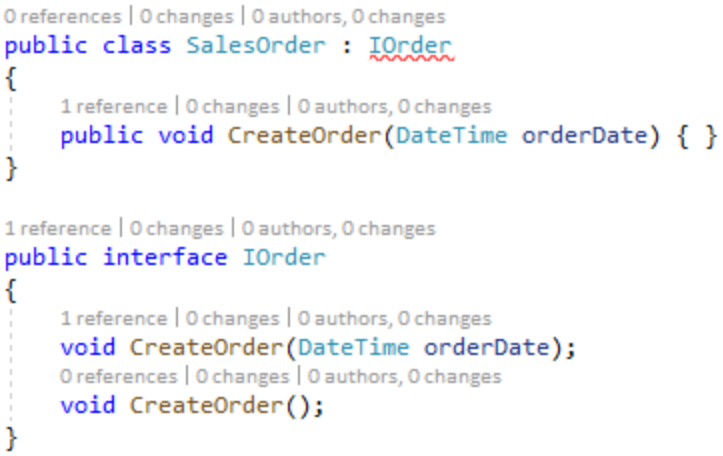

Figure 1: Modifying the interface introduces a breaking change

This is because the addition of the **`CreateOrder()`** method requires implementation in all classes that use the interface. In C# 8.0, however, we can provide a default implementation when upgrading an interface.

Code Listing 31: Default Interface method

```
public class SalesOrder : IOrder
{
    public void CreateOrder(DateTime orderDate) { }
}

public interface IOrder
{
    void CreateOrder(DateTime orderDate);
    void CreateOrder() => CreateOrder(DateTime.Now);
}
```

Now, implementors of the interface that do not know about the new member are not affected. The default implementation is ignored.

Nullable reference types

One of the biggest changes with regards to developer impact is one that probably has the smallest syntactic impact. Developers can now express whether or not a specific reference can be null.

Note: Did you know that null has been around in OOP programming for over 50 years?

The question now is: what happens if the list of students is null? Consider the following code.

Code Listing 32: The ListStudents method

```
private void ListStudents(IEnumerable<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(student.FirstName);
    }
}
```

There is no way we can tell the compiler that the **`Student`** object might be null. With nullable reference types, we can express this more clearly. To enable this feature, you need to add **`<Nullable>enable</Nullable>`** to your .csproj file, as shown in Code Listing 33.

Code Listing 33: Enabling nullable reference types

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

This will now enable more concise compiler feedback to allow you to get the code right the first time. A nullable reference type is indicated by using the same syntax as nullable value types: by adding a **`?`** to the type of the variable, as seen in the method signature of the **`ListStudents`** method in Code Listing 34.

Code Listing 34: Specifying that Student can be null

```
private void ListStudents(IEnumerable<Student?> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(student.FirstName);
    }
}
```

Once you do that, the compiler generates more concise warnings regarding the use of the **`Student`** variable, as seen in Figure 2.

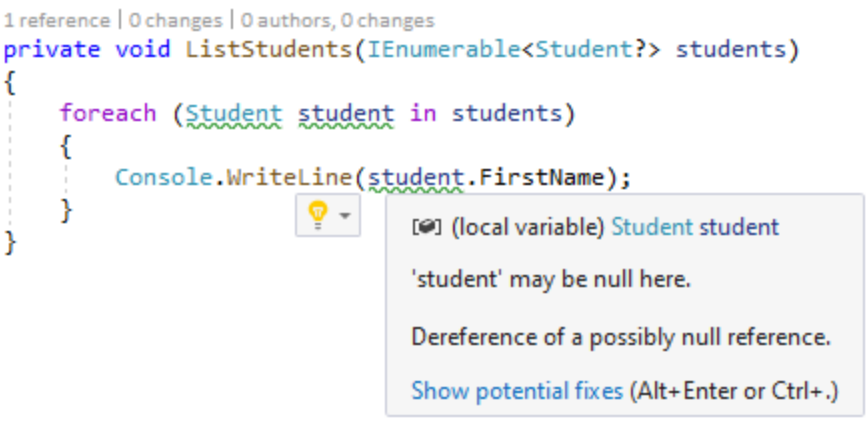

Figure 2: Compiler warnings for null reference

You are now in a better position to code defensively against null reference exceptions.

The null-forgiving operator

Those of you who have worked with the Swift programming language might be familiar with the following syntax. In C# 8.0 we call it the null-forgiving operator, and it is implemented using the **!** operator.

Consider the **Student** class example used in Code Listing 34. To ensure that we have a valid **Student** class, I have created an extension method that ensures my **Student** class is not null, and that the **FirstName** property will have a value. This is illustrated in Code Listing 35.

Code Listing 35: The IsValid extension method

```
public static class ExtensionMethods
{
    public static bool IsValid(this Student student)
    {
        return student != null && !string.IsNullOrEmpty(student.FirstName);
    }
}
```

If I now had to use this in a method that gets the **FirstName** property of the **Student** class, I would see that I still receive the warning when accessing the **FirstName** property.

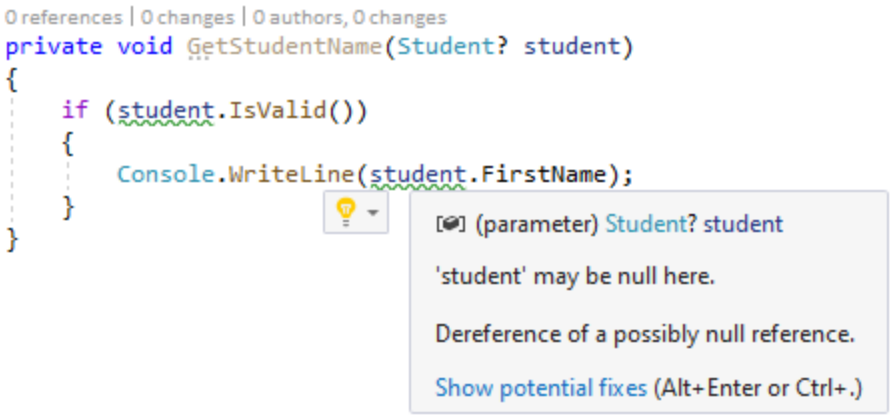

Figure 3: Applying the IsNull extension method

But I am confident that in this instance, because I am calling the **`IsValid`** extension method on my **`Student`** class, the **`FirstName`** property will not be null. I can therefore safely add the null-forgiving operator to my code that reads the **`FirstName`** property, as seen in Code Listing 36.

Code Listing 36: Applying the null-forgiving operator

```
private void GetStudentName(Student? student)
{
    if (student.IsValid())
    {
        Console.WriteLine(student!.FirstName);
    }
}
```

The warning is removed and the intent of my code is quite clear.

Asynchronous streams

The introduction of asynchronous programming has forever changed the way developers write code. With the addition of **`async`** and **`await`** in .NET, C# developers could leverage asynchrony easily. Developers could not, however, consume streams of data asynchronously. That is, not until C# 8.0 introduced **`IAsyncEnumerable<T>`**.

If this looks familiar, that's because it is. **`IAsyncEnumerable<T>`** is similar to **`IEnumerable<T>`**, which is used to iterate over collections. The only difference is that **`IAsyncEnumerable<T>`** allows developers to iterate through a collection asynchronously. This means our code can wait for the next element in a collection without blocking a thread.

Methods that return asynchronous streams have three properties:

- They must be declared with the **`async`** modifier.
- They return an **`IAsyncEnumerable<T>`**.
- They contain **`yield return`** statements to return successive elements in the asynchronous stream.

It is also worth noting that the stream elements are processed in the captured context. To disable this behavior, you need to use the **`TaskAsyncEnumerableExtensions.ConfigureAwait`** extension method.

For more on this, see Microsoft Docs.

To illustrate **`IAsyncEnumerable<T>`**, let us assume that you need to return some data from a data store. It's great if you can get all that data in a single call. You can just perform asynchronous calls to get the data, and return it to the calling code.

The challenge, however, exists when you can't get all that data at once. Sometimes the data needs to be returned in pages as it becomes available.

It is here that asynchronous streams shine. You can now send the data back to the calling code as soon as that data is available. To appreciate **`IAsyncEnumerable<T>`**, let's try to illustrate the problem we are faced with by creating an asynchronous method that mimics the behavior of getting data that is paged.

Consider the following code.

Code Listing 37: Read a stream of data asynchronously

```
static async Task<IEnumerable<int>> GetSomethingAsync()
{
    var iValues = new List<int>();
    for (var i = 0; i <= 10; i++)
    {
        await Task.Delay(1000);
        iValues.Add(i);
    }
    return iValues;
}
```

When you call this method, as shown in Code Listing 38, the application will wait 10 seconds and then display all the numbers at once in the console window.

Code Listing 38: Iterate asynchronously

```
foreach (var item in await GetSomethingAsync())
{
    Console.WriteLine(item);
}
```

To display the numbers on by one as they are generated, let's modify the **GetSomethingAsync** method by changing the **Task<IEnumerable<int>>** to **IAsycEnumerable<int>** and adding **yield**, as seen in Code Listing 39.

Code Listing 39: Using IAsyncEnumerable<T>

```
static async IAsyncEnumerable<int> GetSomethingAsync()
{
    for (var i = 0; i <= 10; i++)
    {
        yield return i;
        await Task.Delay(1000);
    }
}
```

The **yield** keyword performs a stateful iteration and returns the values of a collection one by one. To consume the asynchronous stream, move the **await** keyword before the **foreach** so that your code looks as illustrated in Code Listing 40.

Code Listing 40: Consuming the asynchronous stream

```
await foreach (var item in GetSomethingAsync())
{
    Console.WriteLine(item);
}
```

Running this code will display the numbers in the console window one by one, as they are available.

Asynchronous disposable

If you have a .NET class that makes use of unmanaged resources, then you should see the **IDisposable** interface implemented in that class. This is to allow for the release of unmanaged resources synchronously.

In C# 8.0 you can now do this asynchronously by using **IAsyncDisposable**. This gives you a mechanism for performing resource-intensive dispose operations without blocking the main UI thread.

Having a look at the **IAsyncDisposable.DisposeAsync** method, we see that it returns a **ValueTask** representing the asynchronous dispose operation. You can see more on this here.

When you implement **IAsyncDisposable** on an object in your application, you should call **DisposeAsync** when you are finished using that object. A good practice is to put the code implementing **IAsyncDisposable** in a **using** statement. This ensures that the code releasing the resources will still do so in the event of an exception being thrown.

Indices and ranges

Indices and ranges provide a better and concise way, using bracket notation, to look at a single element from the start or end of an array. They can also be used to look at a range inside of an array.

> ***Note: I use an array as an example here, but it can refer to any sequence of elements.***

Consider the code illustrated in Code Listing 41. (Notice that I am **`using static System.Console`**.)

Code Listing 41: Reading months using indices

```
string[] months =
{                   // From Start       From End
    "January",      // 0                ^12
    "February",     // 1                ^11
    "March",        // 2                ^10
    "April",        // 3                ^9
    "May",          // 4                ^8
    "June",         // 5                ^7
    "July",         // 6                ^6
    "August",       // 7                ^5
    "September",    // 8                ^4
    "October",      // 9                ^3
    "November",     // 10               ^2
    "December"      // 11               ^1
};

WriteLine(months[3]);   // From array start
WriteLine(months[^12]); // From array end
```

We know that C# is zero-based, meaning that the first item in the array starts at 0. Have a look at the **^** operator (some call it the hat operator). Quite controversially, this starts at **`^1`**. The reason for this is that **`^0`** denotes the length of the array. Consider the code illustrated in Code Listing 42.

Code Listing 42: Get items to the end of the array

```
var slice = months[^4..^0];
foreach (var s in slice) WriteLine(s);
```

It reads as follows: get me the months, starting from the fourth element from the end (**`^4`**) for the length (**`^0`**) of the array. Therefore, **`^0`** is one past the end, and points to the very end of the array.

We also see that the code in Code Listing 43 output the same element in the array.

Code Listing 43: Using Length - 1 and ^1

```
WriteLine(months[months.Length - 1]);
WriteLine(months[^1]);
```

You can also pull out a range of values, as illustrated in Code Listing 44.

Code Listing 44: Find a range of values

```
var year = months[..];
foreach (var s in year) WriteLine(s); // January to December

var quarter = months[..3];
foreach (var s in quarter) WriteLine(s); // Quarter 1 - January to March

var restOfYear = months[3..];
foreach (var s in restOfYear) WriteLine(s); // April to December
```

Let's recap some of the rules for indexes:

- Index **`0`** = **`months[0]`**
- Index **`^0`** = **`months.Length`**
- Typing **`months[^n]`** is the same as **`months[months.Length - n]`** where **`n`** is any number.
- A range specifies the start and end of a range with the start of the range being inclusive, and the end of the range being exclusive. Therefore:
    - The range **`months[..3]`** excludes April.
    - The range **`months[3..]`** includes April.
    - The range **`months[0..^0]`** represents January to December.

You can also assign variables, as illustrated in Code Listing 45.

Code Listing 45: Assign variables to index and range

```
var july = ^6;
WriteLine(months[july]); // July

var firstSemester = 0..6;
var semester = months[firstSemester];
foreach (var s in semester) WriteLine(s); // January to June
```

As noted earlier, ranges and indices work with any sequence of elements. You can use them with **`string`**, **`Span<T>`**, or **`ReadOnlySpan<T>`**.

Switch expressions

I have always disliked using **`switch`** statements. Personally, the **`switch`** statement always felt so cumbersome and unnecessarily clunky. Now with C# 8.0, we can be much more concise in the way we express ourselves by using **`switch`** expressions. Consider the traditional **`switch`** statement that returns a string from a method called **`GetBirthStone`**, as illustrated in Code Listing 46.

Code Listing 46: Traditional switch statement

```
private string GetBirthstone(Months month)
{
    switch (month)
    {
        case Months.January:
            return "Ruby or Rose Quartz";
         case Months.March:
             return "Bloodstone and Aquamarine";
         case Months.April:
             return "Diamond";
         case Months.May:
             return "Emerald";
         case Months.June:
             return "Pearl, Alexandrite, and Moonstone";
         case Months.July:
             return "Ruby";
         case Months.August:
             return "Sardonyx and Peridot";
         case Months.September:
             return "Sapphire";
         case Months.October:
             return "Opal and The Tourmaline";
         case Months.November:
             return "Topaz";
         case Months.December:
             return "Turquoise and Zircon";
         default:
             return $"Did not find a birth stone for {month}";
    }
}
```

Compare this with the more concise **`switch`** expression returned from the modified **`GetBirthStone`** method illustrated in Code Listing 47.

Code Listing 47: The new switch expression

```
private string GetBirthstone(Months month) =>
    month switch
    {
```

```
        Months.January   => "Ruby or Rose Quartz",
        Months.March     => "Bloodstone and Aquamarine",
        Months.April     => "Diamond",
        Months.May       => "Emerald",
        Months.June      => "Pearl, Alexandrite, and Moonstone",
        Months.July      => "Ruby",
        Months.August    => "Sardonyx and Peridot",
        Months.September => "Sapphire",
        Months.October   => "Opal and The Tourmaline",
        Months.November  => "Topaz",
        Months.December  => "Turquoise and Zircon",
        _                => $"Did not find a birth stone for {month}",
    };
```

We should note a few things here:

- Because the **GetBirthStone** method just returns the value from the **switch**, it can be changed to use an expression body.
- In the **switch** expression, the need for **case** and **break** keywords are removed.
- The **switch** expression puts the variable **month** before the **switch** keyword.
- The **case** and **:** have been replaced with a single **=>**, which (for me anyway) looks much nicer.
- The **default** case has been replaced with the **_** discard.

This expression body makes for cleaner, better-looking, and more readable code. To change a **switch** statement to a **switch** expression, place your cursor on the **switch** keyword and press **Ctrl+.** and select **Convert switch statement to expression**.

Readonly members

You can now add **readonly** modifiers to **struct** members. This is helpful if you need to indicate that a member does not modify state, and gives you a more fine-tuned approach than simply applying the **readonly** modifier to a struct declaration.

Consider the mutable **struct** in Code Listing 48.

Code Listing 48: Struct to calculate days since a given date

```
public struct DaysSince
{
    public DateTime GivenDate { get; set; }
    public double Number => Math.Round((DateTime.Now -
GivenDate).TotalDays, 0);

    public override string ToString() => $"Days since {GivenDate} =
{Number} days";
}
```

We can see that the **ToString** method will not change the state, and you can indicate this by adding the **readonly** modifier to the **ToString** declaration. When you do this, however, you will receive a compiler warning, as seen in Figure 4.

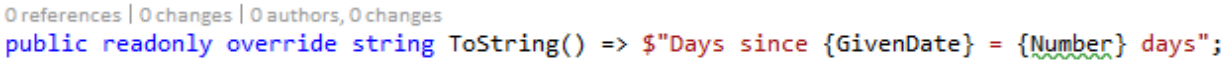

Figure 4: Compiler warning

This happens because the **Number** property is not marked as **readonly**, and the compiler will display this warning when it needs to create a defensive copy. We know that the **Number** property will not change the state, so we can safely add a **readonly** modifier to the declaration.

Code Listing 49: Add readonly modifier to a struct member

```
public struct DaysSince
{
    public DateTime GivenDate { get; set; }
    public readonly double Number => Math.Round((DateTime.Now -
GivenDate).TotalDays, 0);

    public readonly override string ToString() => $"Days since {GivenDate}
= {Number} days";
}
```

Be aware that the **readonly** modifier is only necessary on read-only properties. The compiler will not assume that **get** accessors don't modify state, so you must specify that. The only exception is with auto-implemented properties where all auto-implemented getters are regarded as **readonly** by default. This is the reason that the **GivenDate** property didn't generate a compiler warning.

Using declarations

You should be familiar with the **using** statement in C#. It provides a way to ensure that your code adheres to the correct usage of **IDisposable** objects; when the code execution moves past the **using** statement's scope, the objects in that scope are properly disposed of.

Consider the **using** statement in Code Listing 50.

Code Listing 50: Using statement to read a file

```
private void ReadFile()
{
    using (var reader = new
System.IO.StreamReader("C:\\temp\\TextDocument.txt"))
    {
        var lines = reader.ReadToEnd();
```

```
    }
}
```

With C# 8.0, you can now make use of **`using`** declarations instead, as illustrated in Code Listing 51.

Code Listing 51: Using declaration to read a file

```
private void ReadFile()
{
    using var reader = new
System.IO.StreamReader("C:\\temp\\TextDocument.txt");
    var lines = reader.ReadToEnd();
}
```

What we notice about the **`using`** declaration is that the **`using`** keyword precedes the **`var`** keyword. This tells the compiler that the variable called **`reader`** that is being declared must be disposed of at the end of the enclosing scope.

Static local functions

Local functions are another one of my favorite language features. First appearing in C# 7, you can now add the **`static`** modifier to a local function in C# 8.0. When we see a static local function, we know that it does not use any of the arguments contained in its outer scope. This means that the compiler can optimize the code accordingly.

Consider the following code.

Code Listing 52: Static local functions

```
private double TotalObjectVolume((Cylinder c, Sphere s, Pyramid p)
volumeShapes)
{
    var cylinderVol = CalculateVolume(volumeShapes.c);
    var sphereVol = CalculateVolume(volumeShapes.s);
    var pyramidVol = CalculateVolume(volumeShapes.p);

    return Math.Round(cylinderVol + sphereVol + pyramidVol, 2);

    // static local functions here
    static double CalculateVolume<T>(T volumeShape)
    {
        return volumeShape switch
        {
            Sphere s when s.Radius == 0 => 0,
            Cylinder c  => Math.PI * Math.Pow(c.Radius, 2) * c.Length,
```

```
            Sphere s    => 4 * Math.PI * Math.Pow(s.Radius, 3) / 3,
            Pyramid p   => p.BaseLength * p.BaseWidth * p.Height / 3,
            _ => throw new ArgumentException(message: "Unrecognized
object", paramName: nameof(volumeShape)),
        };
    }
}
```

You will notice that this is the same method we used in the Local Functions demo when discussing C# 7 earlier in the book. The only difference is that now it uses a **switch** expression, as allowed in C# 8.0.

The local function called **CalculateVolume** has been marked as static. The compiler knows that it does not use any of the arguments in the outer scope. To see what this means, add the following static local function to the **TotalObjectVolume** method.

Code Listing 53: Static local function with a compiler error

```
static double GetCylinderRadius()
{
    var cylinder = volumeShapes.c; // Compiler error
    return cylinder.Radius;
}
```

The code for the **TotalObjectVolume** method should now look like Code Listing 54.

Code Listing 54: The TotalObjectVolume method

```
private double TotalObjectVolume((Cylinder c, Sphere s, Pyramid p)
volumeShapes)
{
    var cylinderVol = CalculateVolume(volumeShapes.c);
    var sphereVol = CalculateVolume(volumeShapes.s);
    var pyramidVol = CalculateVolume(volumeShapes.p);

    return Math.Round(cylinderVol + sphereVol + pyramidVol, 2);

    // static local functions here
    static double CalculateVolume<T>(T volumeShape)
    {
        return volumeShape switch
        {
            Sphere s when s.Radius == 0 => 0,
            Cylinder c  => Math.PI * Math.Pow(c.Radius, 2) * c.Length,
            Sphere s    => 4 * Math.PI * Math.Pow(s.Radius, 3) / 3,
            Pyramid p   => p.BaseLength * p.BaseWidth * p.Height / 3,
            _ => throw new ArgumentException(message: "Unrecognized
object", paramName: nameof(volumeShape)),
```

```
        };
    }

    static double GetCylinderRadius()
    {
        var cylinder = volumeShapes.c; // Compiler error
        return cylinder.Radius;
    }
}
```

You will notice that the code we added for the **`GetCylinderRadius`** local function will generate a compiler error, as illustrated in Figure 5.

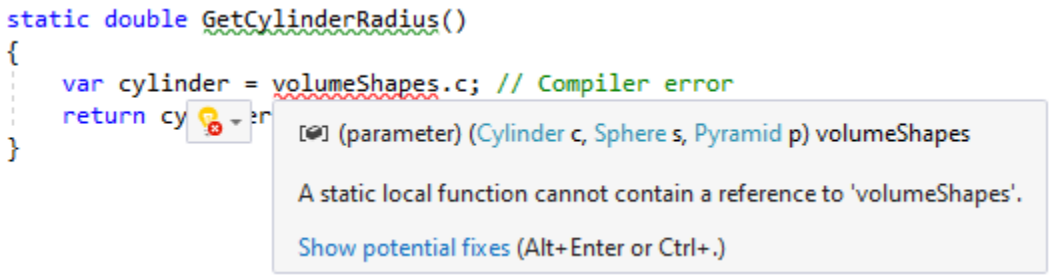

Figure 5: The compiler error on a static local function

This is because the local function is marked as **`static`**, but it references **`volumeShapes`**, which is contained in the outer scope.

Disposable ref structs

In C# 7 we were allowed to declare a **`struct`** with the **`ref`** modifier. What we couldn't do, however, was implement any interfaces on such structs. This means that the code in Code Listing 55 will generate a compiler error.

Code Listing 55: A ref struct implementing an interface

```
ref struct StudentScores : IDisposable
{

}
```

As seen in Figure 6, the compiler is telling us that we can't implement an interface on our struct.

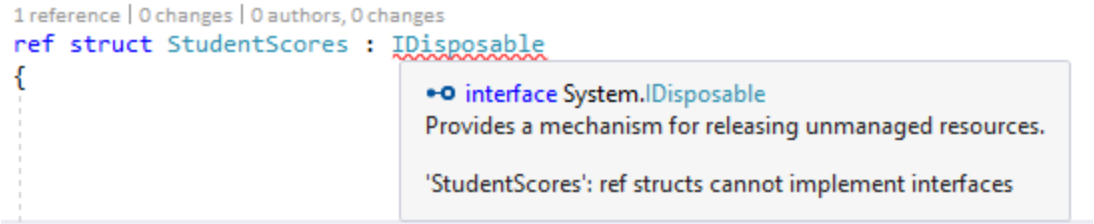

Figure 6: Compiler error on ref struct

This leaves us in a bit of a predicament. What if we needed to perform some cleanup for our struct? In C# 8.0 we can now do just that by adding a publicly accessible **`void Dispose`** method, as seen in Code Listing 56.

Code Listing 56: A ref struct with a Dispose method

```
ref struct StudentScores
{
    public void Dispose()
    {
        // perform clean up
    }
}
```

We can use it in our code in a **`using`** declaration, as illustrated in Code Listing 57.

Code Listing 57: Using declaration for struct

```
using var scores = new StudentScores();
```

You can also use disposable ref structs with **`readonly ref struct`** declarations.

Null-coalescing assignment

C# 8.0 also introduced the null-coalescing assignment operator: **`??=`**. This operator can now be used to assign the value of its right-hand operand to its left-hand operand only in the event of the left-hand operand evaluating to **`null`**.

How often have you seen code like the following?

Code Listing 58: Checking for null and assigning

```
private void AddUpdateScores(List<int> lstScores)
{
    if (lstScores == null)
    {
        lstScores = new List<int>();
```

```
    }

    // Add/Update scores
}
```

In C# 8.0 the null-coalescing assignment operator makes this check almost negligible, as seen in Code Listing 59.

Code Listing 59: Checking for null using null-coalescing assignment

```
private void AddUpdateScores(List<int> lstScores)
{
    lstScores ??= new List<int>();

    // Add/Update scores
}
```

If you read through the code too fast, you might miss it. It's such a small change, but it has quite a big impact. It is also important to note that if the **`lstScores`** variable is not **`null`**; the assignment is simply skipped.

Unmanaged constructed types

Unmanaged types are not types defined in unmanaged code. It is a type that is not a reference type, and does not contain reference type fields at any level of nesting. Therefore, with C# 8.0, a constructed value type is unmanaged if it contains fields of unmanaged types only.

Consider the code in Code Listing 60.

Code Listing 60: A generic struct

```
public struct MyStruct<T>
{
    public T One;
    public T Two;
}
```

In Code Listing 61, we have an extension method with the unmanaged constraint on **`T`**.

Code Listing 61: A generic extension method with an unmanaged constraint

```
public unsafe static PropertyInfo[] GetProps<T>(this T obj) where T :
unmanaged
{
    var t = obj.GetType();
```

```
    return t.GetProperties();
}
```

This means that if we create our struct as illustrated in Code Listing 62, we can call the extension method on the instance of that struct because it is an unmanaged constructed type.

Code Listing 62: Calling the extension method

```
var mystruct = new MyStruct<int> { One = 1, Two = 2 };
var props = mystruct.GetProps();
```

This is because **`int`** is an unmanaged type. If we had to use **`string`**, then we would no longer have an unmanaged constructed type, because **`string`** is not an unmanaged type, and **`mystruct2`** would violate the unmanaged constraint on the extension method.

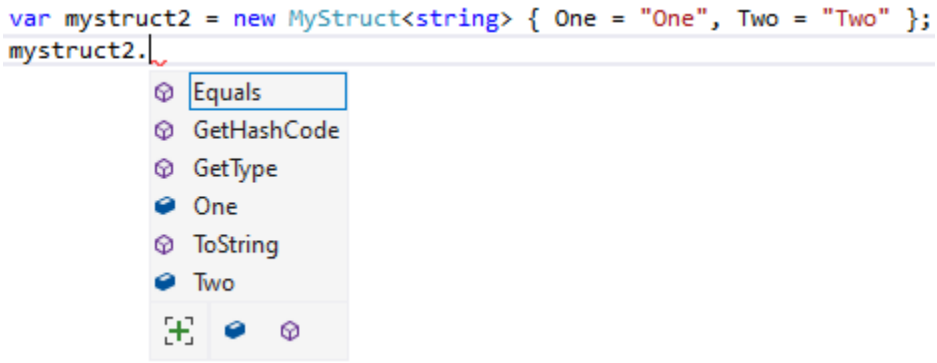

Figure 7: Not-unmanaged constructed type

The following types are unmanaged types:

- **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**, **`char`**, **`float`**, **`double`**, **`decimal`**, or **`bool`**.
- Any **`enum`** type.
- Any pointer type.
- Any user-defined struct type containing only unmanaged type fields and is not a constructed type pre-C# 7.3.

C# 7.3 introduced the unmanaged constraint, as seen in the extension method in Code Listing 61. This means we can use the unmanaged constraint directly in the definition of the generic struct, as seen in Code Listing 63.

Code Listing 63: The unmanaged constraint on the generic struct

```
public struct MyStruct<T> where T : unmanaged
{
    public T One;
    public T Two;
}
```

This would generate a compiler error, as seen in Figure 8, because the creation of **mystruct2** violates the unmanaged constraint on the generic struct definition.

```
var mystruct = new MyStruct<int> { One = 1, Two = 2 };
var props = mystruct.GetProps();

var mystruct2 = new MyStruct<string> { One = "One", Two = "Two" };
```

Figure 8: The MyStruct<string> violates the constraint

We can see that a generic struct can be the source of both unmanaged and not unmanaged constructed types. Where you place the constraint is up to you and what you need to achieve.

Enhancement of interpolated verbatim strings

In C# 8.0 the **$** and **@** tokens used with interpolated strings can be either **$@"..."** or **@$"..."**, and both are now valid interpolated verbatim strings. Before C# 8.0, the **$** token had to appear before the **@** token.

This means **`var msg = $@"The \t student is {studentName}";`** will produce the exact same output as **`var msg = @$"The \t student is {studentName}";`** in the console window.

Enabling C# 8 in any .NET project

It is possible to enable C# 8.0 in any .NET project. There are a few provisos, but I will get to those in a minute. To see this in action, create a .NET console application using the .NET Framework, as seen in Figure 9.

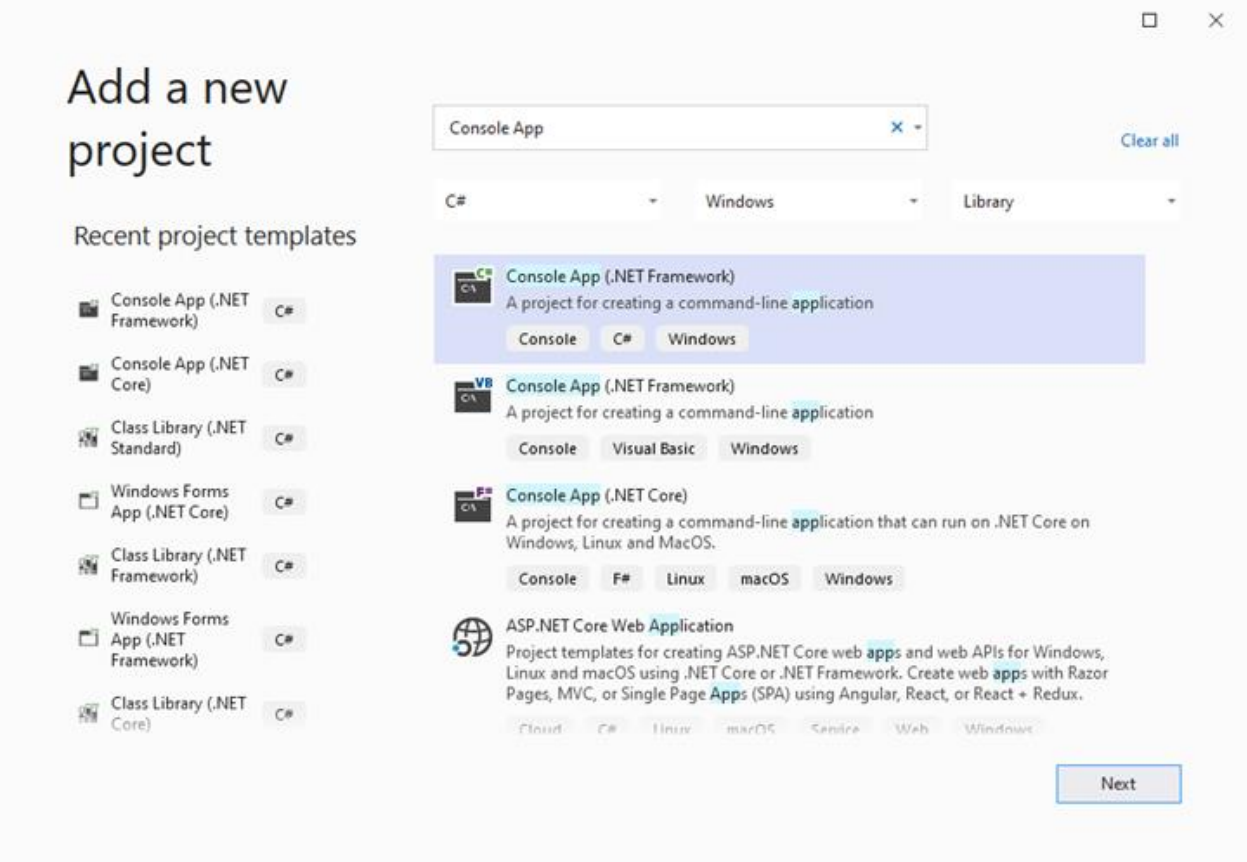

Figure 9: A console app using .NET Framework

When this project is created, add the shape classes seen in Code Listing 64 to your console application. These are the same classes that we used earlier in the book, but I'm presenting them here again for convenience.

Code Listing 64: Shape classes

```
public class Cylinder
{
    public double Length { get; }
    public double Radius { get; }

    public Cylinder(double length, double radius)
    {
        Length = length;
        Radius = radius;
    }
}

public class Sphere
{
    public double Radius { get; }
```

```
    public Sphere(double radius)
    {
        Radius = radius;
    }
}

public class Pyramid
{
    public double BaseLength { get; }
    public double BaseWidth { get; }
    public double Height { get; }

    public Pyramid(double baseLength, double baseWidth, double height)
    {
        BaseLength = baseLength;
        BaseWidth = baseWidth;
        Height = height;
    }
}
```

Let's add our **`TotalObjectVolume`** method to our project that uses a static local function, which also uses a **`switch`** expression. The code is illustrated in Code Listing 65.

Code Listing 65: A static local function using a switch expression

```
private double TotalObjectVolume((Cylinder c, Sphere s, Pyramid p)
volumeShapes)
{
    var cylinderVol = CalculateVolume(volumeShapes.c);
    var sphereVol = CalculateVolume(volumeShapes.s);
    var pyramidVol = CalculateVolume(volumeShapes.p);

    return Math.Round(cylinderVol + sphereVol + pyramidVol, 2);

    // static local functions here
    static double CalculateVolume<T>(T volumeShape)
    {
        return volumeShape switch
        {
            Sphere s when s.Radius == 0 => 0,
            Cylinder c => Math.PI * Math.Pow(c.Radius, 2) * c.Length,
            Sphere s => 4 * Math.PI * Math.Pow(s.Radius, 3) / 3,
            Pyramid p => p.BaseLength * p.BaseWidth * p.Height / 3,
            _ => throw new ArgumentException(message: "Unrecognized
object", paramName: nameof(volumeShape)),
        };
```

```
    }
}
```

At this point, you will see a whole bunch of compiler errors in your static local function. The compiler errors will most likely tell you that you are trying to use C# 8.0 language features in an earlier version of C# (probably C# 7.3, depending on the .NET Framework you are using).

To use C# 8.0 in your console application on the .NET Framework (not .NET Core—remember, we created a regular console app using the .NET Framework), you need to modify your .csproj file as seen in Code Listing 66.

Change the **<LangVersion>** to **8.0** in your .csproj file. If there isn't a **<LangVersion>**, just add one.

Code Listing 66: Add LangVersion to .csproj file

```
<PropertyGroup>
  <Configuration Condition=" '$(Configuration)' == ''
">Debug</Configuration>
  <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
  <ProjectGuid>{16C8CC63-DC92-4D68-BD4E-B10D602777DB}</ProjectGuid>
  <OutputType>Exe</OutputType>
  <RootNamespace>NetFxConsoleApp</RootNamespace>
  <AssemblyName>NetFxConsoleApp</AssemblyName>
  <TargetFrameworkVersion>v4.7.2</TargetFrameworkVersion>
  <LangVersion>8.0</LangVersion>
  <FileAlignment>512</FileAlignment>
  <AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
  <Deterministic>true</Deterministic>
</PropertyGroup>
```

Save your .csproj file and reload your project if needed. That's all there is to it. There are some provisos to this, though.

Not all types are included

Some types, such as **IAsyncEnumerable**, are not included. There is a workaround, though—you can install the Microsoft.Bcl.AsyncInterfaces and Microsoft.Bcl.HashCode NuGet packages.

Indexes and ranges

By using C# 8.0 in non-.NET Core 3 or non-.NET Standard 2.1 projects, you will not be able to use indexes and ranges. This is because these are runtime features, and your code simply will not compile.

Using Directory.Build.props

If you want each project in your solution to target C# 8.0, or if you need to add any custom property to all your projects, you can create a file called **Directory.Build.props** in the root of your solution and add the **`<LangVersion>`** in there.

This is illustrated in Code Listing 67.

Code Listing 67: The Directory.Build.props file

```
<Project>
  <PropertyGroup>
    <LangVersion>8.0</LangVersion>
  </PropertyGroup>
</Project>
```

I am not so sure that I would necessarily enable C# 8.0 on any .NET project. I would probably take the time to create a .NET Core application from the get-go, but this solution isn't without its merits. This could be useful in situations where you are dealing with an existing .NET project that has had a lot of development effort invested, and that can't easily be rewritten from scratch in .NET Core.

In this situation, jimmying the .csproj file to enable C# 8.0 makes sense.

Chapter 4 The Future of C# and C# 9

There are some nice features planned for C# 9. While all these planned features might not make it into the final release—and some things that do make it in might change between the writing of this book and then—they still give a nice overview of the direction the C# team is taking.

Top-level programs

Code Listing 68 illustrates the code that we have all seen, with a **`Program`** class followed by the **`Main`** method.

Code Listing 68: Classes implementing IShape interface

```
using static System.Console;

class Program
{
    static void Main(string[] args)
    {
        var t = new Triangle
        {
            Base = 5.0,
            Height = 10.5
        };

        DisplayArea(t);

        // static local functions
        static void DisplayArea<T>(T shape) where T : IShape
        {
            WriteLine(shape.Area());
        }
    }
}

class Triangle : IShape
{
    public double Height { get; set; }
    public double Base { get; set; }

    public double Area() => Height * Base / 2;
}

class Rectangle : IShape
```

```
{
    public double Length { get; set; }
    public double Width { get; set; }

    public double Area() => Length * Width;

}

public interface IShape
{
    double Area();
}
```

In C# 9, developers will be allowed to simply omit the **Program** class and the **Main** method. The code will look as illustrated in Code Listing 69.

Code Listing 69: Omitting the static void Main

```
using static System.Console;

var t = new Triangle
{
    Base = 5.0,
    Height = 10.5
};

DisplayArea(t);
ReadLine();

// static local functions
static void DisplayArea<T>(T shape) where T : IShape
{
    WriteLine(shape.Area());
}



class Triangle : IShape
{
    public double Height { get; set; }
    public double Base { get; set; }

    public double Area() => Height * Base / 2;
}

class Rectangle : IShape
{
    public double Length { get; set; }
```

```
    public double Width { get; set; }

    public double Area() => Length * Width;

}

public interface IShape
{
    double Area();
}
```

This means you can just write top-level statements at the top of your file. The top-level statements remain part of your **Main** method, and the **DisplayArea** local function remains a local function; it's just called from the top-level statements.

Top-level statements must precede namespaces and type declarations, and can only appear in one file.

Another great feature is that if you place your **await** statements as top-level statements, then the **Main** becomes an **async Task Main**. At this point, I bet you are wondering what will become of the **args** argument in the **Main** method.

This is not in any of the previews just yet, but there is a possibility that **args** will become a magic keyword. This means the C# team will make a variable available in the top-level statements called **args**. The magic variable makes me think of **value** in a property setter.

Relational and logical patterns

C# 9.0 will introduce patterns that correspond to relational operators, such as **<** and **<=**. You will also be able to combine patterns with logical operators such as **and**, **or**, and **not**. Consider the code in Code Listing 70.

Code Listing 70: Method with switch expression

```
public AreaSize DoSomething<T>(T shape, int numberOfShapes) where T :
IShape
{
    var area = shape switch
    {
        Square s => s.Area() * numberOfShapes,
        Circle c => c.Area() * numberOfShapes,
        null => throw new ArgumentNullException(nameof(shape)),
        _ => throw new ArgumentException(message: $"Unknown shape:
{shape}", paramName: nameof(shape))
    };

    if (area < 3.0)
```

```
        return AreaSize.small;
    else if (area < 5.0)
        return AreaSize.medium;
    else if (area < 7.0)
        return AreaSize.large;
    else
        return AreaSize.huge;
}
```

We have checked for null in the **switch** expression, and if the **shape** is null, we throw an **ArgumentNullException**. We can move the **null =>** to the top of the **switch** expression if we want to (because if it is **null**, why carry on?). The position here doesn't matter, but what is clear is that if we reach the discard **_ =>**, we know that the shape is not **null**.

Consider the code in Code Listing 71. At this point in the **switch** expression, we can be certain that **shape** is not null.

Code Listing 71: The discard in the switch expression

```
_ => throw new ArgumentException(message: $"Unknown shape: {shape}",
paramName: nameof(shape))
```

This means we can make our intent clearer by using the **not** logical operator, as illustrated in Code Listing 72.

Code Listing 72: Using the not logical operator

```
not null => throw new ArgumentException(message: $"Unknown shape: {shape}",
paramName: nameof(shape))
```

Secondly, because C# 9.0 will add relational patterns, we can modify the **if**/**else** statement, as illustrated in Code Listing 73.

Code Listing 73: The if else statement to convert

```
if (area < 3.0)
    return AreaSize.small;
else if (area < 5.0)
    return AreaSize.medium;
else if (area < 7.0)
    return AreaSize.large;
else
    return AreaSize.huge;
```

Interestingly enough, Visual Studio 2019 version 16.7.0 Preview 4.0 supports converting this statement to a **switch** expression using relational patterns, as seen in Figure 10.

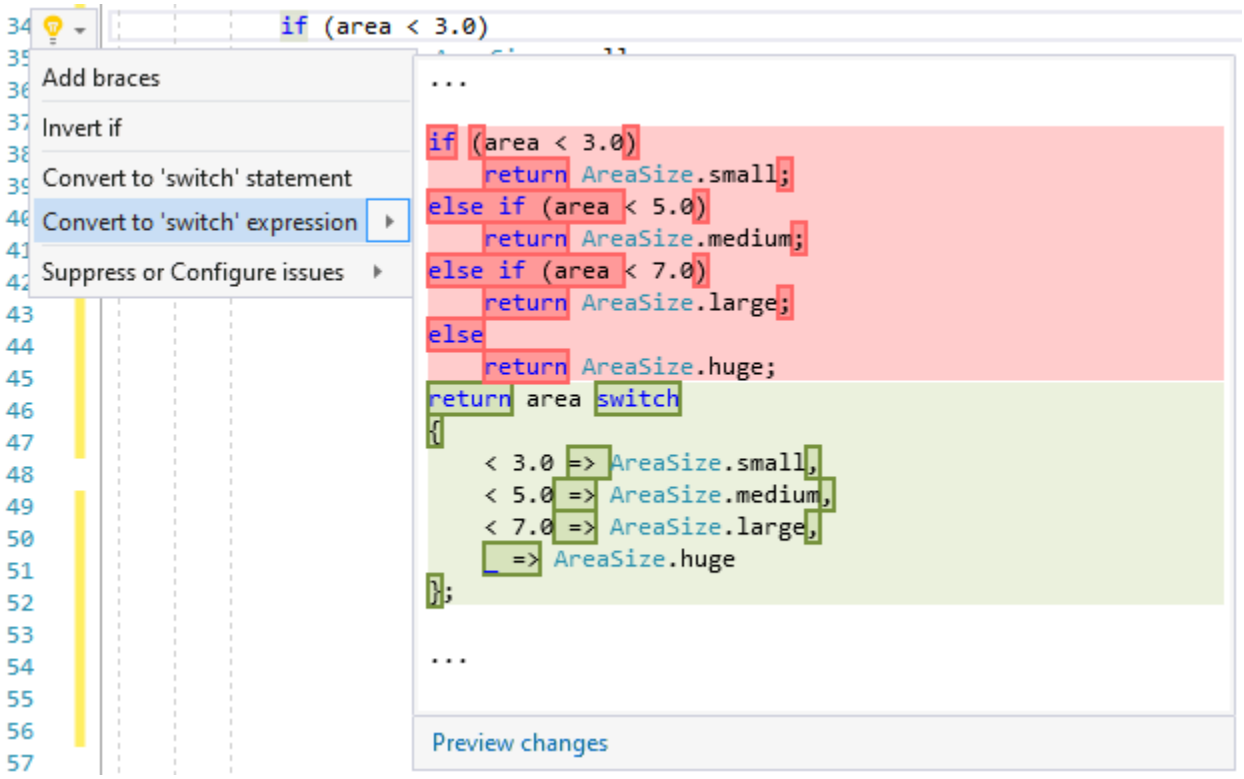

Figure 10: Convert to switch expression

The converted **`switch`** expression looks nice and succinct, as illustrated in Code Listing 74.

Code Listing 74: A switch expression using a relational pattern

```
public AreaSize DoSomething<T>(T shape, int numberOfShapes) where T :
IShape
{
    var area = shape switch
    {
        Square s => s.Area() * numberOfShapes,
        Circle c => c.Area() * numberOfShapes,
        null => throw new ArgumentNullException(nameof(shape)),
        not null => throw new ArgumentException(message: $"Unknown shape:
{shape}", paramName: nameof(shape))
    };

    //Relational pattern(min 21)
    return area switch
    {
        < 3.0 => AreaSize.small,
        < 5.0 => AreaSize.medium,
        < 7.0 => AreaSize.large,
        _     => AreaSize.huge
```

```
    };
}
```

Taking logical patterns further, we can now get rid of unwieldy double parentheses for **`if`** conditions. Consider the code in Code Listing 75.

Code Listing 75: Standard if not condition

```
if (!(shape is Circle)) { }
```

We can now replace this with the code in Code Listing 76.

Code Listing 76: New if not condition using a logical pattern

```
if (shape is not Circle) { }
```

If a shape can be a **`Circle`** or a **`Square`**, we can do the following:

Code Listing 77: Logical or pattern

```
if (shape is Circle or Square) { }
```

This will give developers a fantastic way to express the intent of their code clearly and succinctly.

Target-typed new expressions

Before C# 9.0, whenever you wrote a **`new`** expression in C#, you were required to specify the type. The only exception was implicitly typed arrays, where you would create the following array:

Code Listing 78: Implicitly typed array

```
var planets = new[] { "Mars", "Saturn", "Jupiter" };
```

In C# 9.0, you will be allowed to omit the type when it's clear what type the expression is being assigned to. Consider the following code.

Code Listing 79: The new expression for creating a Circle

```
Circle c = new Circle(5);
```

In C# 9.0, this code can simply be written as follows.

Code Listing 80: The target-typed new expression

```
Circle c = new (5);
```

This code looks neater and conveys exactly what it should. I do, however, think that those of us using **var** will probably not use target-typed **new** expressions too often.

Init-only properties

C# allows developers to use object initialization, which is a very convenient and flexible way of creating a new object. Consider the **SalesOrder** class in Code Listing 81. It currently uses an auto-property for **OrderNumber**.

Code Listing 81: The SalesOrder class

```
public class SalesOrder
{
    public string OrderNumber { get; set; }
}
```

One limitation is that the properties have to be mutable for object initializers to work. The object's default, parameterless constructor is called, and then the property is assigned. But you can set the property to a different value after initialization because it is mutable, as seen in Code Listing 82.

Code Listing 82: Initializing the SalesOrder class

```
var salesOrder = new SalesOrder
{
    OrderNumber = "123"
};

salesOrder.OrderNumber = "345";
```

With init-only properties, this mutability is fixed. The **init** accessor is a variant of the **set** accessor, and can only be called during object initialization. If you modify your property to use **init**, as seen in Code Listing 83, then any subsequent assignments to the **OrderNumber** property will result in a compile-time error.

Code Listing 83: Setting init-only property on SalesOrder class

```
public class SalesOrder
{
    public string OrderNumber { get; init; }
}
```

Visual Studio will tell you that your assignment after object initialization is not allowed.

```
var salesOrder = new SalesOrder
{
    OrderNumber = "123"
};

salesOrder.OrderNumber = "345";
```

Figure 11: Subsequent assignment results in a compile-time error

This is because the **`OrderNumber`** property is not mutable.

Init accessors and readonly fields

Consider the same **`SalesOrder`** class we had a look at earlier. Modifying it as illustrated in Code Listing 84, you will notice the following.

Code Listing 84: Init accessor on the readonly field

```
public class SalesOrder
{
    private readonly string orderNumber;
    public string OrderNumber
    {
        get => orderNumber;
        init => orderNumber = (value ?? throw new
ArgumentNullException(nameof(OrderNumber)));
    }
}
```

This is possible because **`init`** accessors can only be called during initialization. This means that they can mutate **`readonly`** fields in the enclosing class.

Records

In the previous code listings, we saw that we can make individual properties immutable by using the **`init`** accessor. As seen in Code Listing 85, we can make the whole **`SalesOrder`** class become immutable and behave like a value by adding the **`data`** keyword.

Code Listing 85: Creating a record

```
public data class SalesOrder
{
```

```
    public string OrderNumber { get; init; }
}
```

Adding the **data** keyword to the class declaration marks the class as a record. This means records are seen more as values, and less as objects—they don't have a mutable encapsulated state. To represent any change over time, you must create a new record that represents the new state, meaning they are defined by their contents.

More C# 9.0 goodies

These are only some of the planned features for C# 9.0. While I know a lot could change before its release, the code illustrated in the previous examples gives us a nice glimpse of where the C# team is headed. If you want to keep up to date on what is happening around C# 9.0, swing over to the Language Feature Status page on GitHub. This allows you to see planned C# 9.0 features and their current states.

Chapter 5 .NET Productivity Features in Visual Studio

There are a variety of nice productivity features in Visual Studio that can enhance your workflow and make your day-to-day tasks easier. Currently, I am using Visual Studio 2019 version 16.7 preview 5.0, but some of these features are also available in the current release of Visual Studio 2019.

Developer PowerShell inside Visual Studio

Inside Visual Studio, if you press **Ctrl**+`, you will open the Visual Studio terminal.

> ***Tip: The ` or backtick is located on the same key as ~ on the keyboard. It is also the same keystroke as used in Visual Studio Code. This makes Ctrl+` a very convenient, one-handed keystroke used to open the terminal inside Visual Studio.***

Launching the terminal opens the integrated Developer PowerShell instance, as seen in Figure 12.

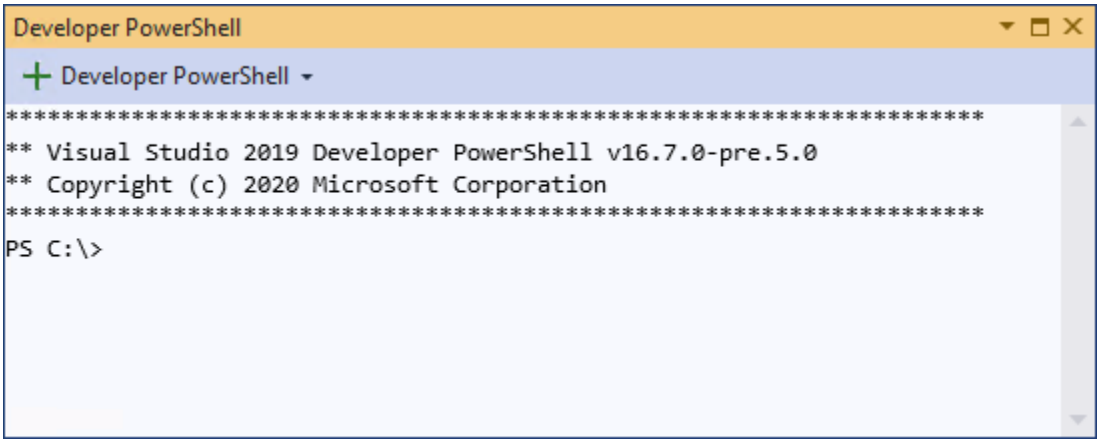

Figure 12: Developer PowerShell

If you are not comfortable using PowerShell, you can switch to using the Developer Command Prompt, as seen in Figure 13.

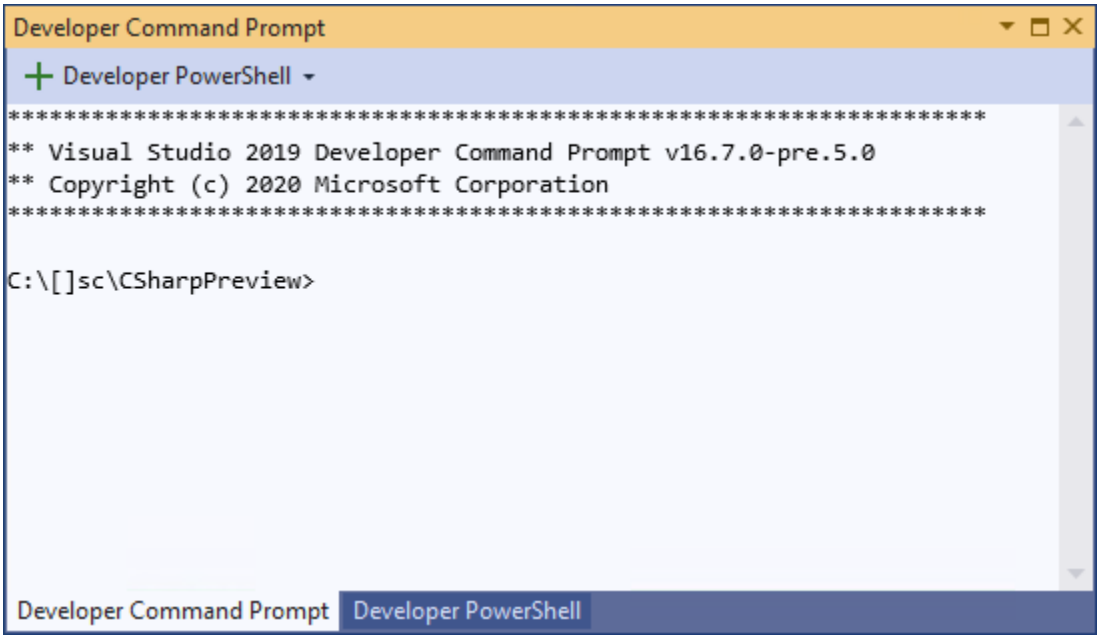

Figure 13: Developer Command Prompt

You can easily select whichever terminal you want from the drop-down, as seen in Figure 14.

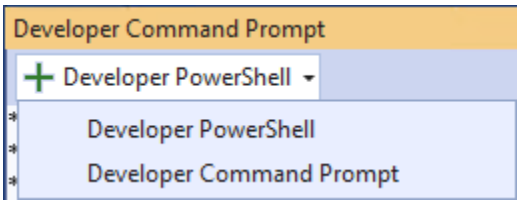

Figure 14: Selecting different terminals

You no longer need to leave Visual Studio to use the terminal window. This is definitely a time-saver.

The Visual Studio Git Window

Visual Studio will now include a new Git window. This is currently available in the Preview edition of Visual Studio 2019.

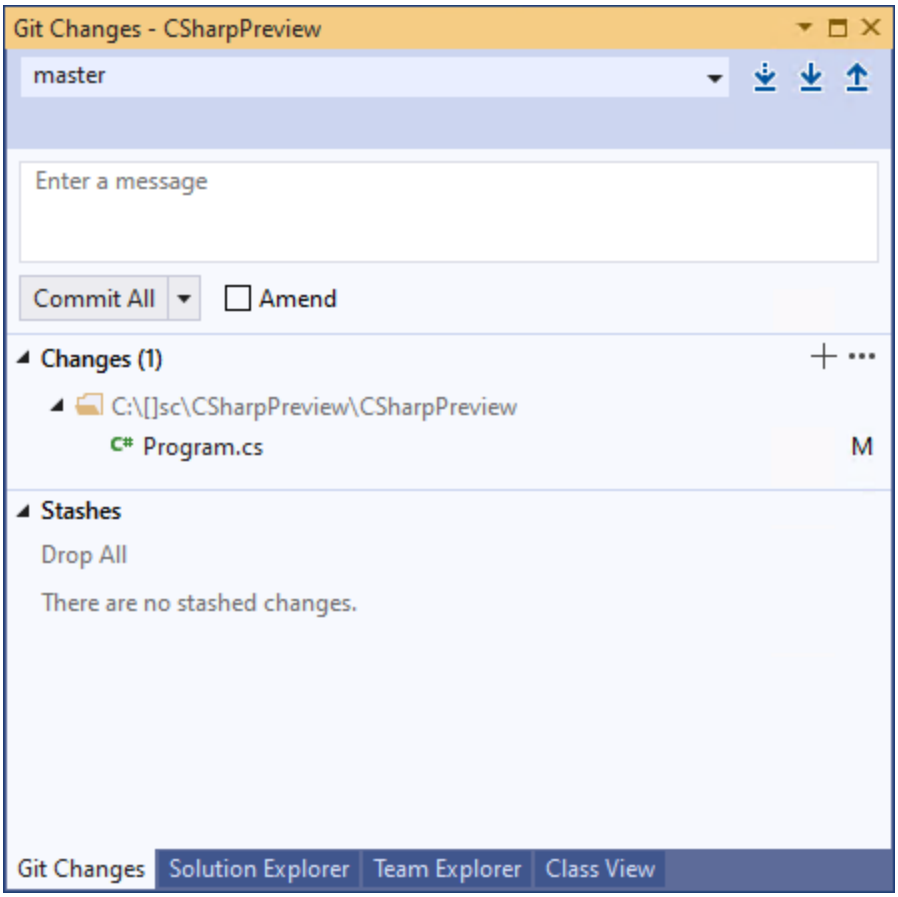

Figure 15: The new Git window

As seen in Figure 15, the new Git window is laid out rather nicely, and contains everything you would expect to see.

You can click the branch (currently set to **master** in Figure 16) and create a new branch.

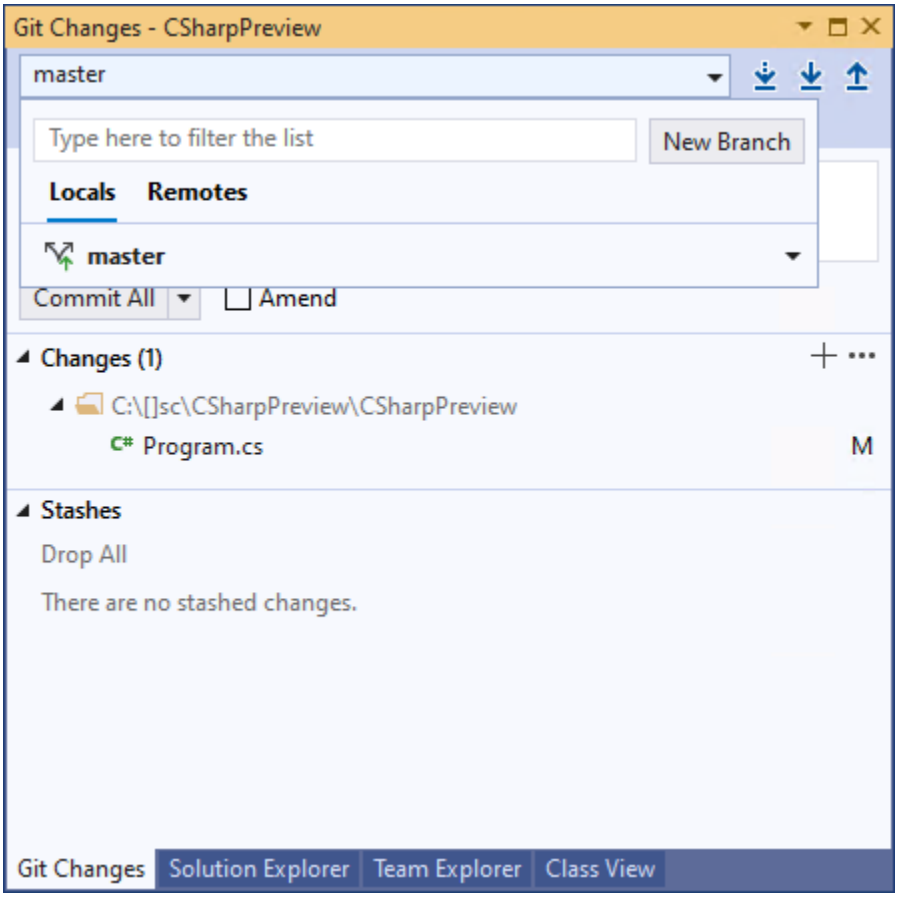

Figure 16: Managing branches

This will display the window in Figure 17. Here you can specify a branch name and select the branch you want to base your new branch on.

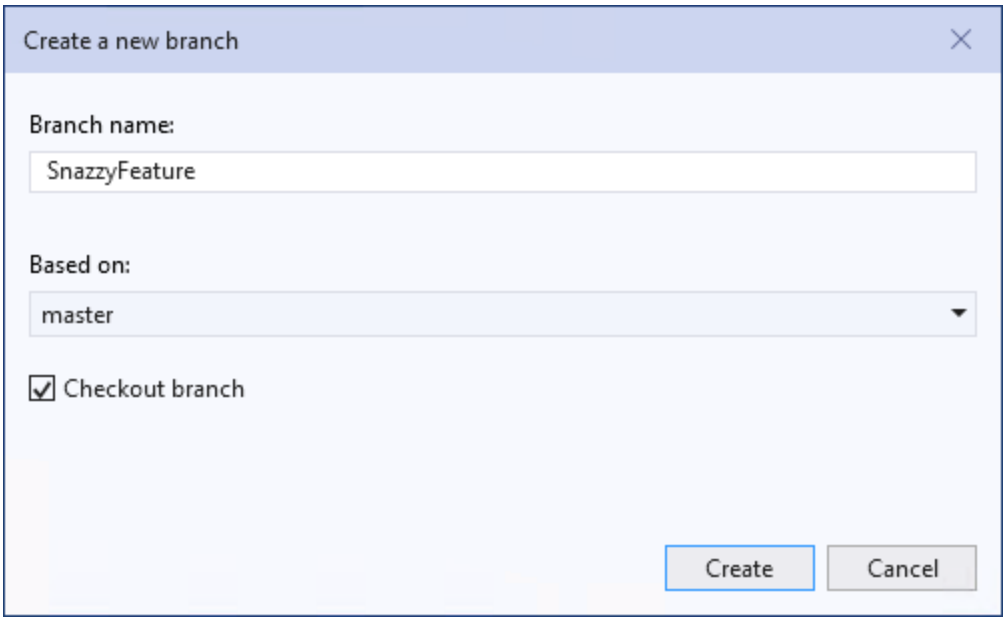

Figure 17: Creating a new branch

Once the new branch is created, it is available under the branches drop-down (Figure 18).

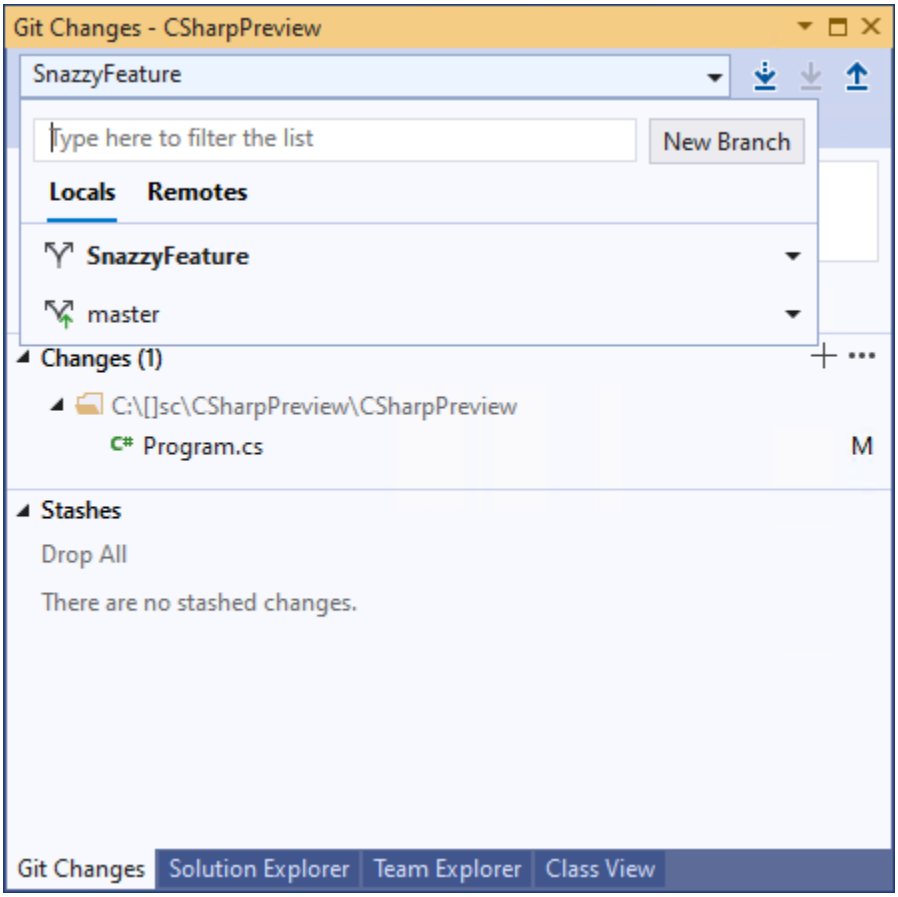

Figure 18: Branches in the Git window

Managing your branch is also easily done from the drop-down next to the branch name, as seen in Figure 19.

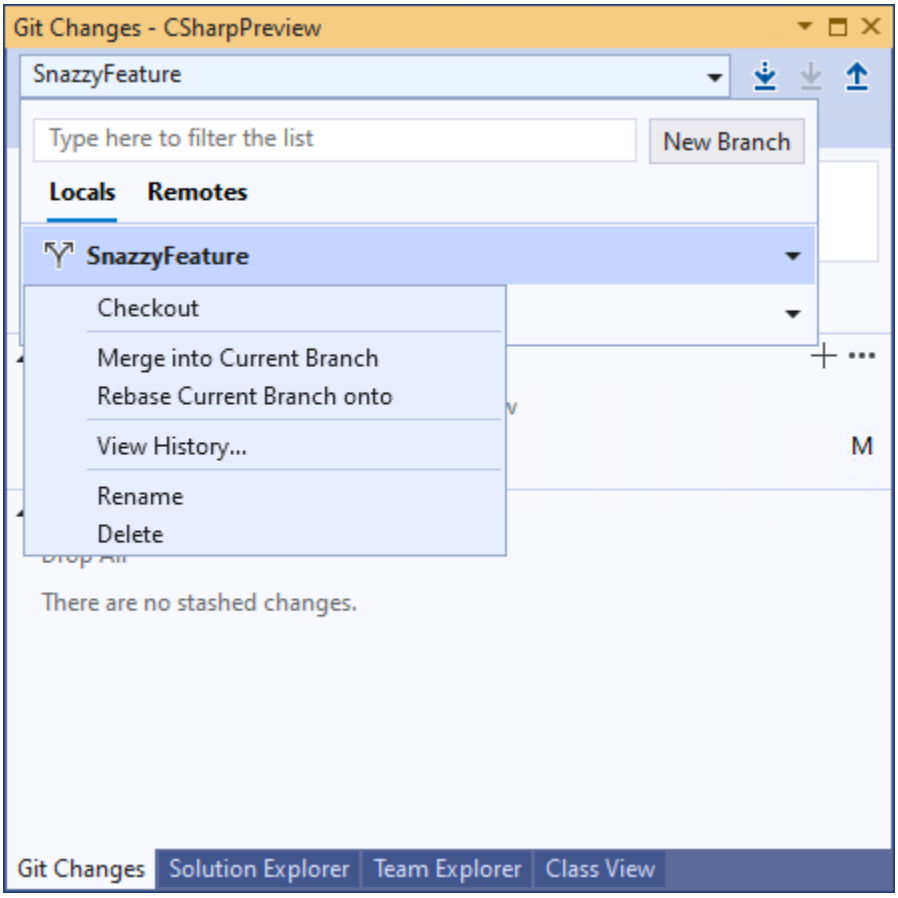

Figure 19: Managing your branch

It is here that you can checkout, merge, rebase, rename, delete, or view the branch history.

Double-clicking a file will show you the code changes in the diff view right inside your editor, as seen in Figure 20.

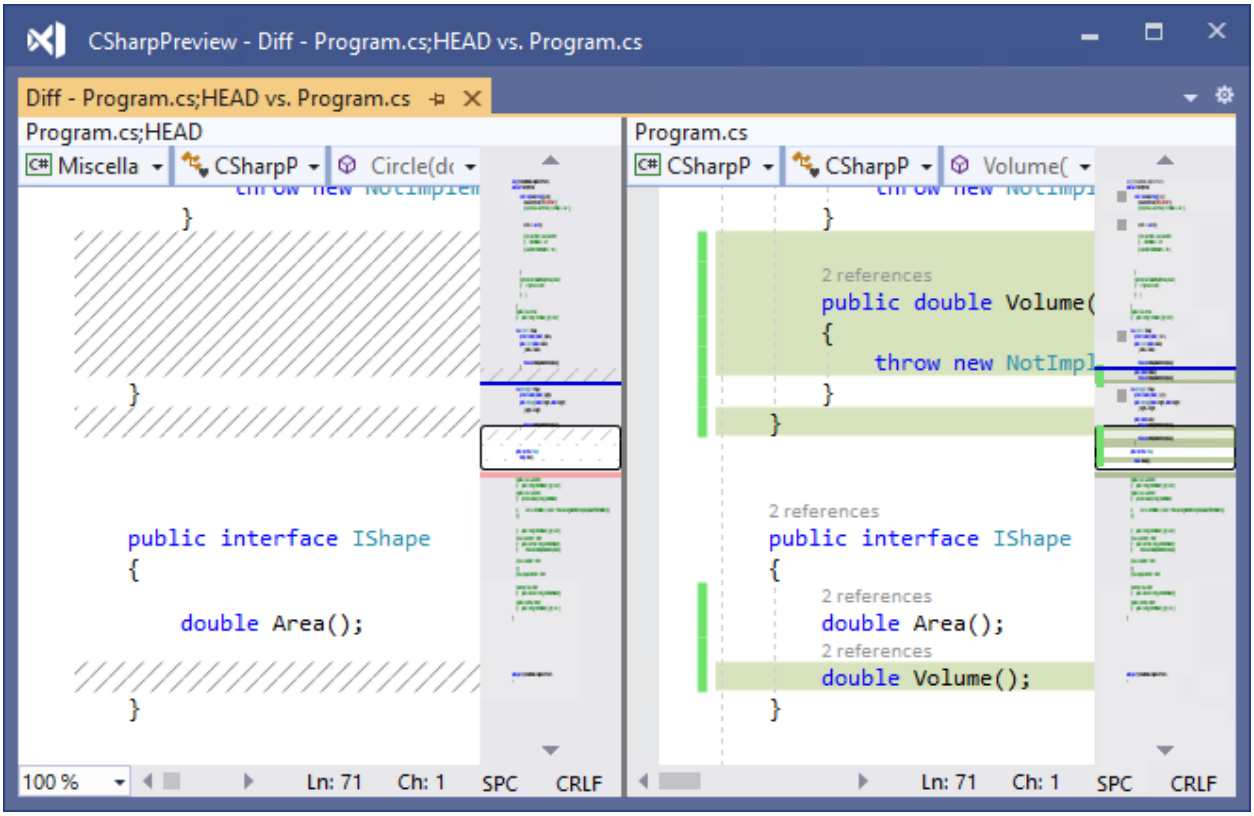

Figure 20: View code changes

The Git window also allows you to do the usual tasks such as stashing, staging, undoing changes, fetching, pulling, pushing, and syncing. According to Mika Dumont, a Program Manager on .NET and Visual Studio, the purpose of the Git window was to streamline the Git experience as the Team Explorer user interface was becoming somewhat busy.

Drag and drop projects to add a reference

In Figure 21, you will notice that I have two projects in the solution. These are:

- CSharpPreview
- CSharpPreview.Core

If I want to add a reference to **CSharpPreview.Core** from my **CSharpPreview** project, I can simply click and drag the **CSharpPreview.Core** project onto the **CSharpPreview** project.

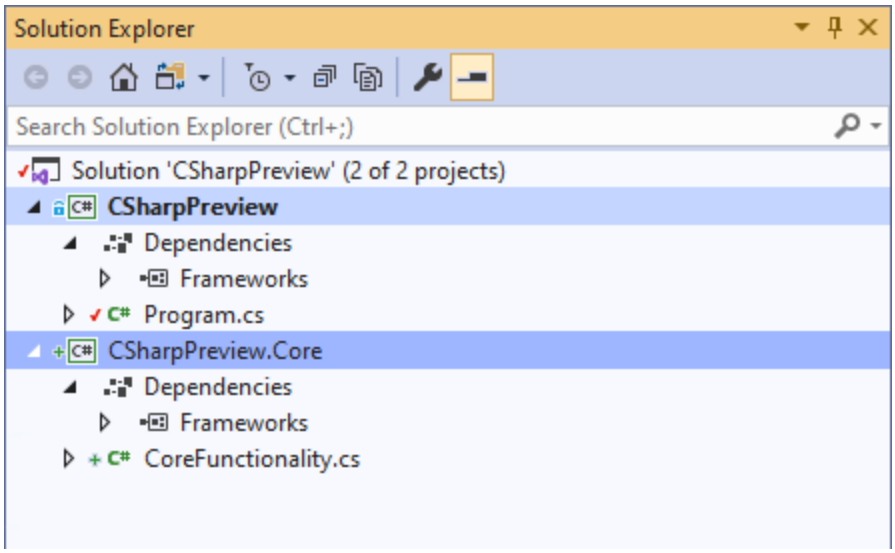

Figure 21: Adding a project reference

A reference to **CSharpPreview.Core** will be added, as seen in Figure 22.

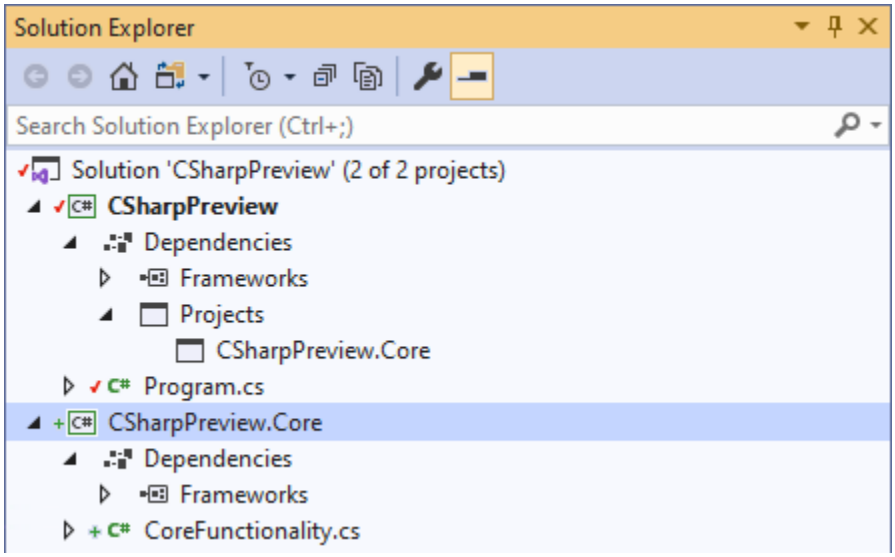

Figure 22: Project reference added

There's no more right-clicking and adding a reference. You can also drag a file from File Explorer to the Solution Explorer and drop it inside a project. The file will then get copied to your source. Nice!

Searching Visual Studio

Another nice feature in Visual Studio is **Ctrl**+**Q**, a great keyboard shortcut that gives the Search text box in Visual Studio focus, allowing you to start typing immediately (Figure 23).

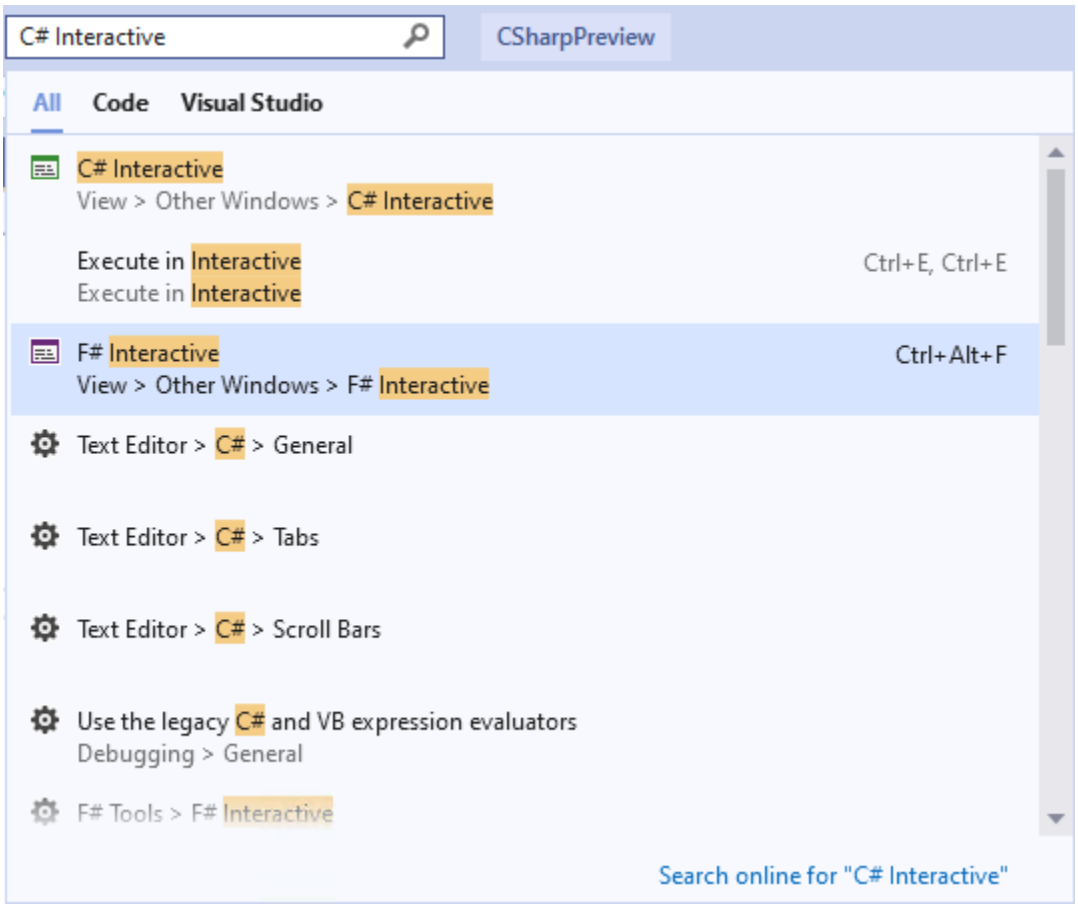

Figure 23: Search Visual Studio

You can further filter your search results by only viewing results in Visual Studio or results in your code (Figure 24).

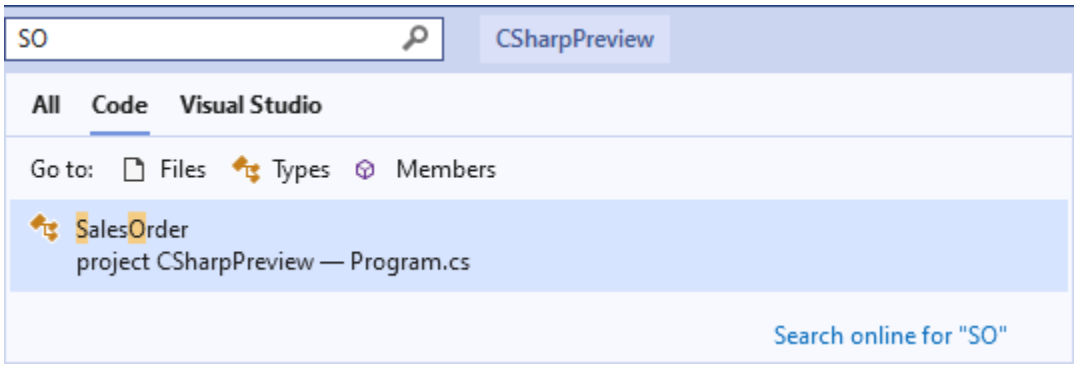

Figure 24: Searching code via camel case

Interestingly enough, you can search your code in Visual Studio by only typing in camel case, as seen in Figure 24. This is great, especially if someone in the team decided to create a **`SalesOrderFunctionalPreProcessor`** class.

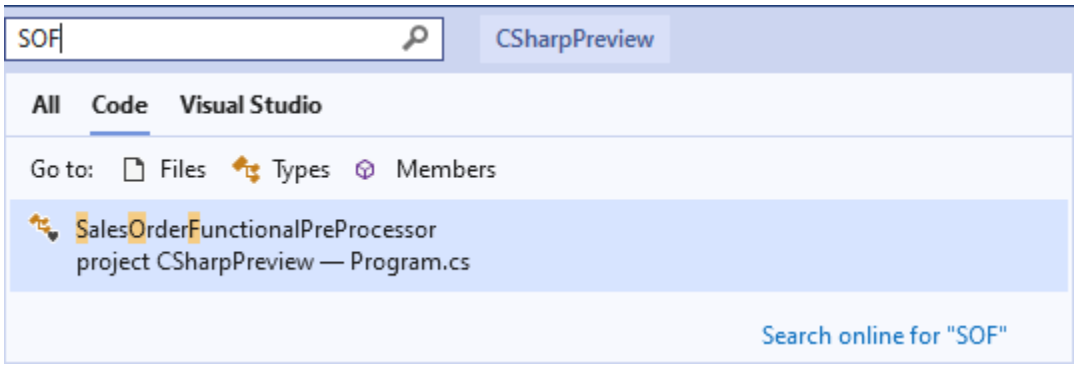

Figure 25: Partial camel case character search

Being able to type in **SOF** in this instance (Figure 25), makes searching code quick and easy.

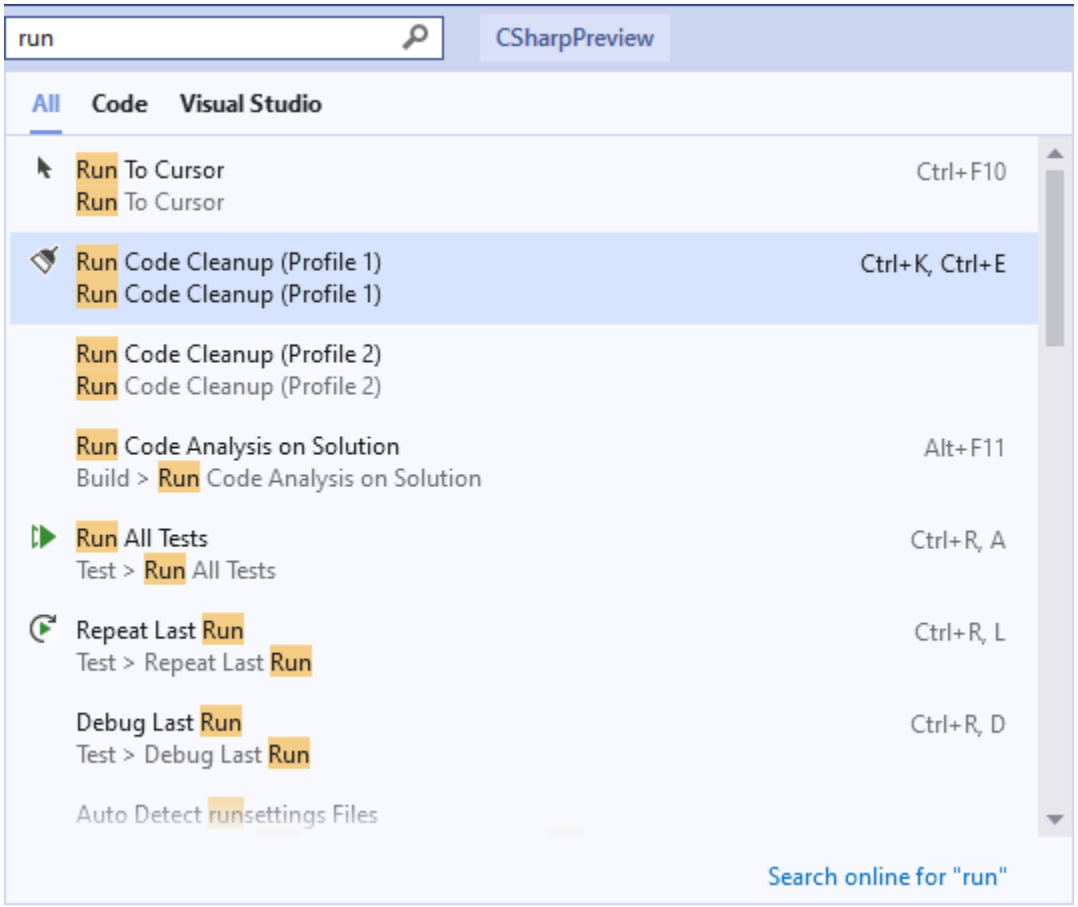

Figure 26: Search commands to run

The search also allows you to run commands (Figure 26). This enables you to run tests, for example, and includes the keyboard shortcut, which is super helpful.

Code analyzers

Code analyzers ensure that your code is more readable, and they will give you suggestions as you go along. Consider Figure 27.

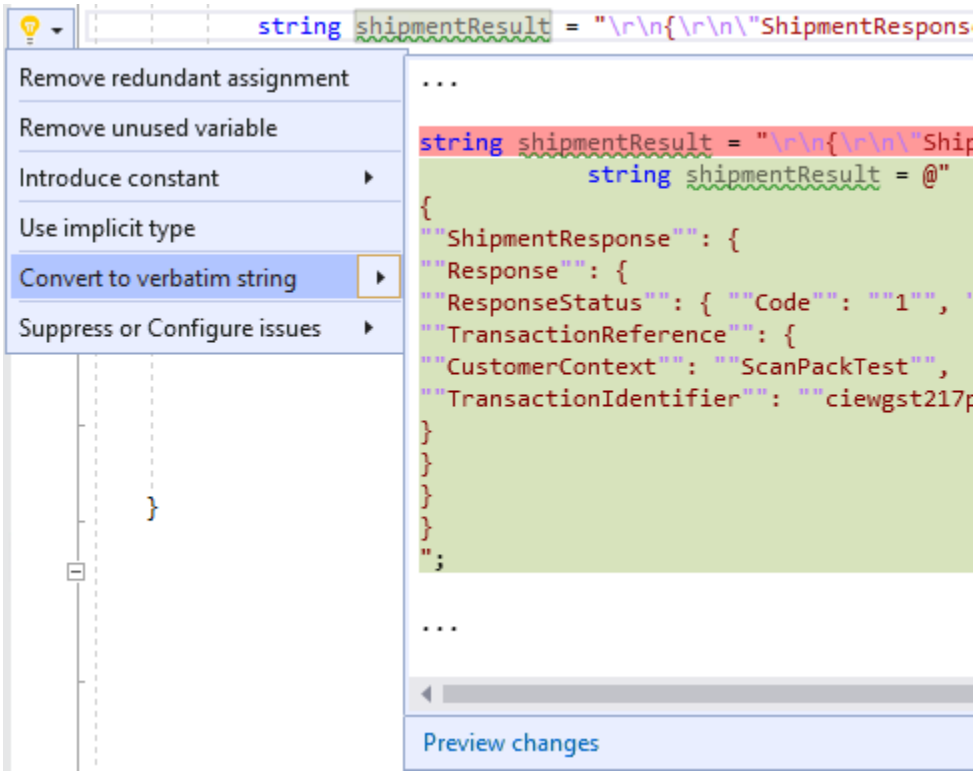

Figure 27: Lightbulb code suggestion

The lightbulb will suggest improvements to your code as you write it. Some of these suggestions can, however, be easily missed. Consider Figure 28.

```
var factorCount = intFactors.Count > 0 ? intFactors[intFactors.Count - 1] : 0;
```

Figure 28: Code fix suggestion

If you look carefully at **`intFactors.Count - 1`**, you will notice the ellipsis (three dots) below the “in” portion of the variable **`intFactors`**.

If you click on the ellipsis, you will see the code fix suggestion (Figure 29). In this example, the suggestion is to use the index operator. Have a look at Code Listing 41 for more on the index operator.

Figure 29: Use index operator suggestion

You can also change the severity of this code fix suggestion by changing the severity from the same place you applied the code fix (Figure 30).

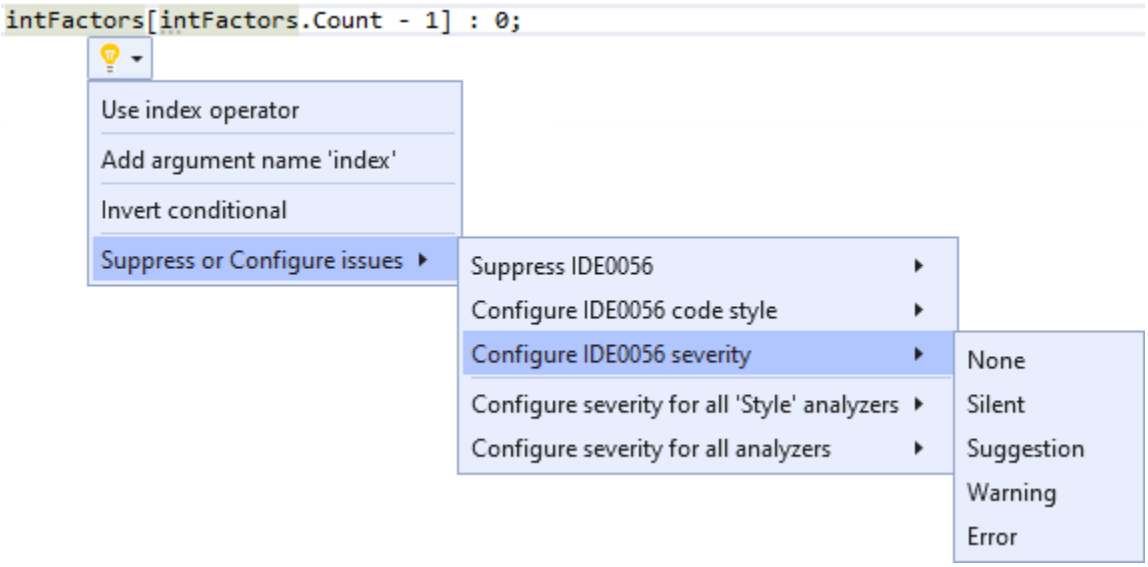

Figure 30: Configure severity

Changing this code fix to a warning will give you a squiggly line under the code being analyzed, as seen in Figure 31.

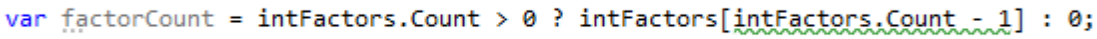

Figure 31: Severity promoted to warning

Code analyzers are a fantastic feature of Visual Studio, and they help developers and teams maintain good coding practices.

File header support in .editorconfig

The .editorconfig file is a universal way to enforce a specific code style or code quality options across your team (Figure 32).

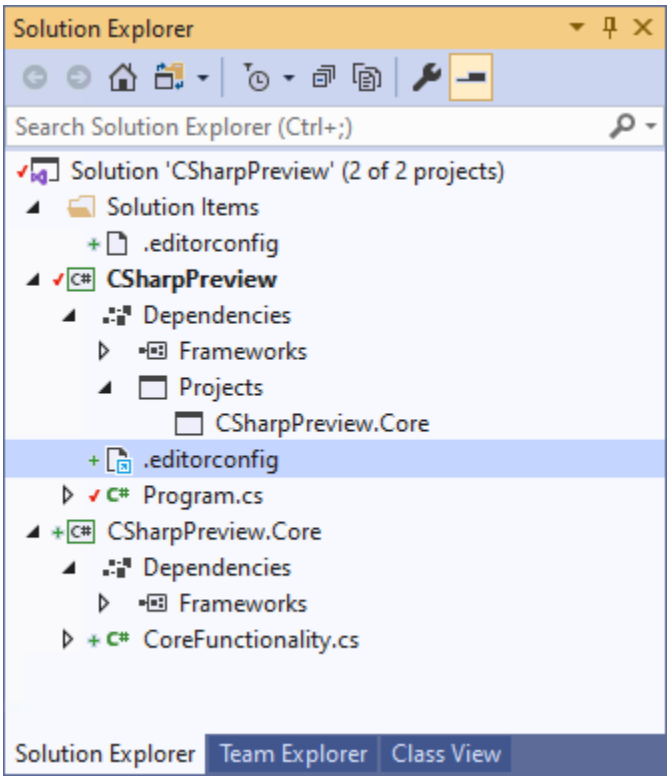

Figure 32: The .editorconfig file

This can be checked into source control and can travel with your solution to new team members.

If you open the .editorconfig file (Figure 33), you will notice the warning we added for the index operator in Figure 30. The .editorconfig now includes file header support. I can now add file headers to files in my solutions using my .editorconfig file. To do this, add the text you want to the **`file_header_template`** as seen in Figure 33, and save your .editorconfig.

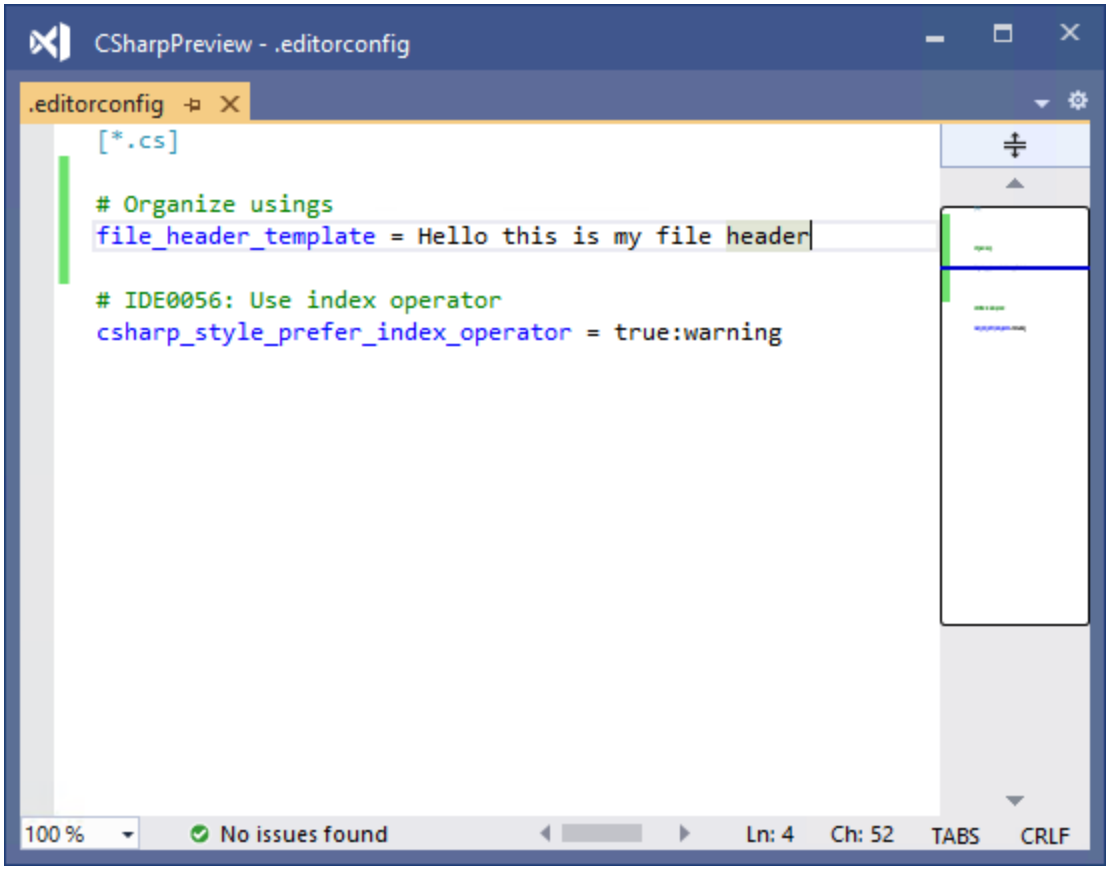

Figure 33: Modifying the .editorconfig file

In a code file, place your cursor at the very top before the first **`using`** statement, and click the lightbulb that is displayed (Figure 34).

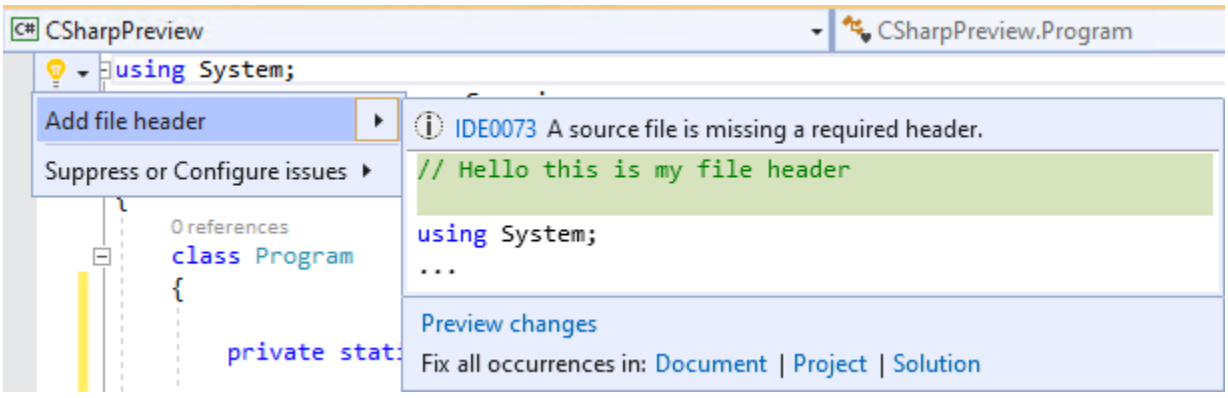

Figure 34: Add file header

You are now able to include this file header in your code file. You can also choose to add it to the document, project, or solution.

C# language resources

Microsoft has made many resources available to developers. There is a rich body of code online, as well as various communities that allow developers to find help if they need it. Here are a few resources you might find useful.

C# language reference

This Introduction document is probably the one you will be reading the most.

C# language proposals

Language proposals should be seen as living documents, and would be the place to go to see what the current thinking is surrounding a specific language feature.

C# language design meetings

The language design meetings (also known as LDM) is where the C# team investigates, designs, and decides on features being added to the C# language. You can read the meeting notes here.

C# language design

This is the official repo for the C# language.