Bonus Chapter 6: C# 8 Updates

What's In This Chapter?

Code Downloads for This Chapter

The code downloads for this chapter are found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the `CSharp8` folder. The code for this chapter is divided into the following major examples:

- NullableSample
- UsingDeclarationSample
- ReadonlyMembersSample
- DefaultInterfaceMembersSample
- AsyncStreamsSample
- SwitchStateSample
- RangesSample

Language Improvements

C# is continuously evolving. C# 8 is released, and C# 9 is on the way. Future proposals and meeting notes are discussed in the open – on https://github.com/dotnet/csharplang.

C# 8 has many improvements to enhance productivity and reduce errors, which is shown in this bonus chapter.

Enabling C# 8

C# 8 is the default version of the programming language creating .NET Core 3.0/3.1 applications and .NET Standard 2.1 libraries. For other application types, you need to enable it. With previous versions of Visual Studio, it was possible to change the C# version with the Advanced Build options in the project settings. This is no longer the case. Now you need to change the project file (file extension `.csproj`).

There's a good reason for this behavior change. With new C# versions so far, there have been some issues with switching the compiler version. No Microsoft does it in a way similar to other frameworks – there's no Microsoft support in using new C# versions with older frameworks. You can still decide on your own to switch to new C# versions with older framework versions you still need to support. However, not all the new features are working everywhere. You will not have issues with features that are just syntax sugar. Nonetheless, features that require a new runtime do not work. *Default interface members* belong to this group. This feature requires a change in the .NET runtime which is only implemented with .NET Core 3.0 and above and is available only with .NET Standard 2.1 libraries. Other features are based on new types such as indices and ranges require the `Index` and `Range` type to be available. Pay attention to where you use the new C# version, and what features you use in case you change the configuration to use C# 8 with older frameworks.

Project Settings

With a C# project file, you can directly specify the C# language version:

```
<LangVersion>8.0</LangVersion>
```

Instead of specifying the specific language version, you can also define to use the latest released C# version with the value `latest`, or the installed preview version with the value `preview`.

Default Configurations

Instead of specifying the language version with every project, you can also define the setting with the file `Directory.Build.props`. This file is searched for in every base directory.

The following file (configuration file `CSharp8/Directory.Build.props`) defines the 8.0 language version as well as enabling the nullable C# 8 features for all the projects found in directories within this directory:

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

Enabling and Disabling Nullability

The C# 8 feature for nullable reference types is not enabled by default. With existing projects, you can get a huge number of compilation warnings, that's why this setting is disabled by default. Another reason the setting is not enabled by default is that you get all the advantages of this feature is when the libraries are written with this feature enabled. A goal is that all the major NuGet packages should have this feature enabled first, and later on, this feature should be enabled by application creators.

However, with new applications, you can turn this feature already on without big hassles. With existing applications, this feature can be turned on as well, and you can get rid of the compiler warnings over time by turning this feature on and off with different sections in the code.

To change nullable features in the code, you can use the nullable preprocessor directive.

```
#nullable disable
```

Disables nullability. In case you've enabled nullability you can disable it in the current file scope.

```
#nullable restore
```

With `#nullable restore`, the setting is back as it was before, e.g. with the project configuration.

```
#nullable enable
```

With `#nullable enable`, nullability is enabled, no matter how the project is configured.

`type="note"`

Note

If you have a big number of compiler warnings because of nullability you can turn nullability off with the project scope and turn it on from file to file

where you fix nullability using `#nullable enable` and `#nullable restore`. If turning it on only leads to a small number of errors, leave it on with the project settings, and turn it off with file scope as needed.

Nullability

Why is that buzz around nullability? What is this all about? It's probably the most important feature of C# 8: Nullable Reference Types. With C# 7, you can assign null to any reference type. This is no longer the case with C# 8 – if nullable reference types are enabled as shown in the previous section. This change is a breaking change with existing code, that's why this feature needs to be explicitly enabled.

What's the reason for this change? The number 1 exception with .NET applications is the `NullReferenceException` – an exception happening when members of a reference that is null is accessed. With .NET guidelines, no applications should throw exceptions of type `NullReferenceException`. Instead, if receiving parameters with null values, a method should check for that and throw a `ArgumentNullException` instead. The `ArgumentNullException` exception has the advantage that the errors are thrown where needed, not in some unexcepted behavior. `NullReferenceException` exceptions are not that easy to detect.

`type="warning"`

Note

From the guidelines https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/using-standard-exception-types#nullreferenceexception-indexoutofrangeexception-and-accessviolationexception: DO NOT allow publicly callable APIs to explicitly or implicitly throw `NullReferenceException`, `AccessViolationException`, or `IndexOutOfRangeException`. These exceptions are reserved and thrown by the execution engine and in most cases indicate a bug. Do argument checking to avoid throwing these exceptions. Throwing these exceptions exposes implementation details of your method that might change over time.

To avoid `NullReferenceException`, previous versions of C# already introduced some features, such as the *null coalescing operator*, and the *null conditional operator* (also known as *null propagation operator*). C# 8 goes

bit steps further. Enabling nullability, you cannot assign null to reference types – only to reference types that are declared as nullable types.

type="reference"

Reference

The null coalescing and null conditional operators are covered in Chapter 6, “Operators and Casts.”

The new nullability is easy to understand. The syntax is similar to nullable value types. Just like for value types, if the `?` is not specified with the declaration of the type, `null` is not allowed:

```
int i1 = 4; // null is not allowed
int? i2 = null; // null is allowed

string s1 = "a string"; // null is not allowed
string? s2 = null; // a nullable string
```

While the syntax with value types and reference types now looks similar, the functionality behind the scenes is very different.

- With value types, the C# compiler makes use of the type `Nullable<T>`. This type is a value type and adds a Boolean field to define if the value type is null or non-null.
- With reference types, the C# compiler adds the `Nullable` attribute. Version 8 of the compiler knows about this attribute and behaves accordingly. C# 7 and older versions of C# do not know about this attribute, and just ignore it.

Compiling a program with C# 8, both `Book b` and `Book? b` becomes `Book b` with C# 7. With this, a library built with C# 8 can still be used with C# 7. However, with a C# 7 compiler you do not get the advantages of methods declared with nullability. Instead, you can pass null values where null is not allowed, and best get an `ArgumentNullException` (if the arguments are verified), or a `NullReferenceException` in other cases.

Let’s get into an example. The following `Book` class defines the non-nullable properties `Isbn` and `Title`, and the nullable property `Publisher`. In addition to that, this type contains a constructor making use of the C# 7 tuples as well as deconstruction implementing the Deconstruct method. Using the `Book` type and accessing the `Publisher` property, a value can only be written to a variable of type `string?`. Assigning it to `string` results in the C# compilation warning “*converting null literal or possible null value to a non-nullable type*” (code file `CSharp8/NullableSample/Book.cs`):

```
public class Book
```

```
{
  public string Isbn { get; }
  public string Title { get ; }
  public string? Publisher { get; }
  public Book(string isbn, string title, string? publisher = default) =>
    (Isbn, Title, Publisher) = (isbn, title, publisher);

  public void Deconstruct(out string isbn, out string title,
                          out string? publisher) =>
    (isbn, title, publisher) = (Isbn, Title, Publisher);

  public override string ToString() => Title;
}
```

Assigning Nullable to Non-Nullable

In case you need to assign a nullable type (like the `Publisher` property from the `Book` type), the C# compiler analyzes the code. Directly assigning a nullable value to a non-nullable value results in a compilation error. Using the `if` statement to verify for not null, the compiler knows the value is not null, and it can be assigned to the non-nullable type without warning. Using the *null coalescing operator* is another option where a value can be specified to assign to in case the expression on the left is null. Of course, it's also possible to assign a nullable to a nullable reference as shown with the variable `publisher4` (code file `CSharp8/NullableSample/Program.cs`):

```
static void AssignNullableToNonNullable(Book b)
{
  // string publisher1 = b.Publisher; // compiler warning
  if (b.Publisher != null)
  {
    string publisher2 = b.Publisher; // ok
  }

  string publisher3 = b.Publisher ?? string.Empty;
  string? publisher4 = b.Publisher;
}
```

Invoking Methods

Before invoking a method (or accessing any member) of a nullable reference type, you need to verify if the reference is not null. This can be done by doing a check beforehand, or by using the *null conditional operator* (code file `CSharp8/NullableSample/Program.cs`):

```
static void ShowBook(Book book)
{
  Console.WriteLine($"{book.Title.ToUpper()} {book.Publisher?.ToUpper()}");
}
```

Declaring Parameters and Return Types

To define if null should be allowed with a parameter, declare the parameter type name with a question mark at the end. This allows to pass null to this method. The method GetPublisher1 allows to pass null as a parameter. With this the implementation should be fine with null and needs to be implemented accordingly. In the code snippet, the book parameter is checked with a null conditional operator, and an empty string is returned if null is passed. If the book received is not null, the value of the Publisher property is returned. If the `Publisher` property itself returns null, the null conditional operator comes into play again, and an empty string is returned:

```
public string GetPublisher1(Book? book) => book?.Publisher ?? string.Empty;
```

To define that null can be returned, the `?` is used with the return type. With the method `GetPublisher2`, the `Book` parameter might not be null. However, the Publisher property is declared to be nullable, and the method `GetPublisher2` can return null - or a publisher:

```
public string? GetPublisher2(Book book) => book.Publisher;
```

Initialization in the Constructor

Using nullability, some code changes may be necessary. For example, to initialize a non-nullable member reference, a helper method called from the constructor could be seen in existing code such as invoking the `RegisterServices` method from within the constructor to assign the `Services` property. The code as shown here results in this compiler warning: “The non-nullable property Services is uninitialized. Consider declaring the property as nullable.” (code file `CSharp8/NullableSample/ConstructorSample1.cs`):

```
public class AppServicesLegacy
{
  public AppServicesLegacy()
  {
    RegisterServices();
  }

  public void RegisterServices()
  {
    var services = new ServiceCollection();
    //...
    Services = services.BuildServiceProvider();
  }

  public IServiceProvider Services { get; private set; }
}
```

Instead of declaring the `Services` property as a nullable type, a better change would be to directly assign the `Services` property in the constructor, and to return a reference from the `RegisterServices` method. This way, the private set accessor can also be removed from the `Services` property as it's only assigned within the constructor (code file `CSharp8/NullableSample/ConstructorSample2.cs`):

```
public class AppServices
{
  public AppServices()
  {
    Services = RegisterServices();
  }

  public IServiceProvider RegisterServices()
  {
    var services = new ServiceCollection();
    //...
    return services.BuildServiceProvider();
  }

  public IServiceProvider Services { get; }
}
```

Null Forgiving Operator

In case you need to override warnings by the compiler, e.g. where the returned value is not null, but there are some reasons you can't write the code as needed, you can use the **null forgiving operator**.

type="note"

Note

Before the release of C# 8, the *null forgiving operator* was known as *dammit operator*.

Let's get into an example declaring a `BooksContext` for Entity Framework Core. Here the `BooksContext` class derives from the class `DbContext`, and defines a property `Books` of type `DbSet<Book>` for mapping the `Book` type to the Books table. With the code as shown here, and nullability enabled, you get a compiler warning Non-nullable property `Books` is uninitialized. The initialization is done by the base class which is not checked by the compiler. One way to deal with the compiler warning is to declare the property of type `DbSet<Book>?`. This way it's ok that this property has a null value, and the compiler does not complain. However, you need to verify for not null every time this property is used.

```
public class BooksContext : DbContext
{
  public BooksContext(DbContextOptions<BooksContext> options)
```

```
    : base(options)
  {
  }

  public DbSet<Book> Books { get; set; }
}
```

This can be resolved by assigning a value to the property on construction of the object. Assigning null (using the default keyword), leads to another issue because null cannot be assigned to non nullable references. Using `!` (the **null forgiving operator**) you tell the compiler to ignore this issue and allow this without complaining with a compiler warning (code file `CSharp8/NullableSample/BooksContext.cs`):

```
public class BooksContext : DbContext
{
  public BooksContext(DbContextOptions<BooksContext> options)
    : base(options)
  {
  }

  public DbSet<Book> Books { get; set; } = default!;
}
```

Null Coalescing Assignments

Another feature related with nullability is **null coalescing assignments**. Instead of first checking for null, and then assigning a value, using the null coalescing assignment operator, this can be done at once.

Let's have a look at an example with different implementations. In the first example, the `GetBook1` method checks if the variable `_book` is null using the `if` statement. If the variable is null, a new instance is created and assigned to the _book variable. Then the book is returned from the method (code file `NullableSampleApp/Program.cs`):

```
private Book? _book;
public Book GetBook1()
{
  if (_book == null)
  {
    _book = new Book("4711", "Professional C# 9");
  }
  return _book;
}
```

With the next iteration, the same variable is used, but instead of using the if statement, the null coalescing operator is used. If the result on the left side is null, the right side is called where a new book is assigned to the _book variable, and finally this value is returned (code file `NullableSampleApp/Program.cs`):

```
private Book? _book;
public Book GetBook()
{
  return _book ?? (_book = new Book("4711", "Professional C# 9"));
}
```

The null coalescing operator is not new with C# 8, but the *null coalescing assignment operator* is. Here, the check for null and the assignment of a new object (if the result on the left side is null), is combined into the `??=` operator (code file `NullableSampleApp/Program.cs`):

```
private Book? _book;
public Book GetBook()
{
  return _book ??= new Book("4711", "Professional C# 9");
}
```

Small Features

Before getting into the some more big features of C# 8 let's look into some small and easy go grasp features that can still improve productivity.

Using declaration

C# 8 has a new feature for the `using` keyword. In this book, using was already used as the *using directive* to open namespaces. The *using statement* is used to free up resources by calling the `Dispose` method of the `IDisposable` interface. C# 8 now gives us the *using declaration*. The using declaration invokes the `Dispose` method like the using statement but with a simplified syntax.

type="reference"

Reference

The using directive is covered in Chapter 2, "Core C#". The using statement is covered in detail in Chapter 17, "Managed and unmanaged memory".

First, let's start with a class implementing `IDiposable` - the `AResource` class (code file `CSharp8/UsingDeclarationSample/AResource.cs`):

```
public class AResource : IDisposable
{
  private bool disposedValue;

  public void Use()
```

```
  {
    Console.WriteLine("use the resource");
  }

  protected virtual void Dispose(bool disposing)
  {
    if (!disposedValue)
    {
      if (disposing)
      {
        // TODO: dispose managed state (managed objects)
      }

      // TODO: free unmanaged resources (unmanaged objects)
      // and override finalizer
      // TODO: set large fields to null
      disposedValue = true;
    }
  }

  public void Dispose()
  {
    Dispose(disposing: true);
    GC.SuppressFinalize(this);
  }
}
```

Using this class with a traditional using statement creates try/finally code and invokes the `Dispose` method at the end of the `using` block (code file `CSharp8/UsingDeclarationSample/Program.cs`):

```
private static void UsingStatement()
{
  using (var r = new AResource())
  {
    r.Use();
  }
}
```

A new using declaration is started without using parentheses as with the using statement and requires a variable. With the using statement a variable is not necessary - you can for example invoke a method that returns an `IDiposable` object, but do not assign it to a variable. With the using declaration, a variable is required. The using declaration creates try/finally code and invokes the `Dispose` method at the end of the scope of the variable. Here, the scope of the variable is the method, thus the resource is disposed at the end of the method (code file `CSharp8/UsingDeclarationSample/Program.cs`):

```
private static void UsingDeclaration()
{
  using var r = new AResource();
  r.Use();
}
```

The using declaration reduces indentation. This comes very practical when multiple resources need to be disposed.

In case you need to create a shorter scope for disposal before the end of the method, curly brackets can be used to define a shorter lifetime scope for the variable.

Pattern-based using

Another feature of C# 8 is pattern-based using. A ref struct cannot implement interfaces. In this case, just the `Dispose` method is enough to use this type with the using declaration or the using statement. For all other types it's still necessary to implement the interface `IDiposable`.

type="reference"

Reference

The ref struct is explained in Chapter 17, "Managed and Unmanaged Memory."

The following code snippet demonstrates a ref struct implementing the Dispose method - which is enough for this type to be used with a using declaration (code file `UsingDeclarationSample/ARefStructResource.cs`):

```
public ref struct ARefStructResource
{
  public void Foo() => Console.WriteLine("Foo");
  public void Dispose()
  {
    Console.WriteLine($"ARefStructResource:Dipose");
  }
}
```

Static local functions

Local functions (functions within a method, property accessor, event accessor…) can now be declared with the static modifier. This way it's made sure that the local function cannot access any instance member or any variable in the scope of the method - outside of the local function. The compiler can also optimize the code because initialization is not needed.

type="reference"

Reference

Local functions are explained in Chapter 13, "Functional Programming with C#."

The following sample shows a local function as implemented in an extension class. Here, a local function is used to throw the

`ArgumentNullException` when the `Where1` method is invoked from the calling code, and not at a later time when the iteration is done (e.g. using a `foreach` loop). The local function with the name `Iterator` is defined within the `Where1` method and can only be used within the scope of this method. The `Iterator` local function accesses variables outside of this function, such as the `source` and the `pred` variable. This is not allowed in a static local function (code file `StaticLocalFunctionsSample/CollectionExtensions.cs`) :

```
public static IEnumerable<T> Where1<T>(
  this IEnumerable<T> source, Func<T, bool> pred)
{
  if (source == null) throw new ArgumentNullException(nameof(source));
  if (pred == null) throw new ArgumentNullException(nameof(pred));

  return Iterator();

  IEnumerable<T> Iterator()
  {
    foreach (T item in source)
    {
      if (pred(item))
        yield return item;
    }
  }
}
```

The local function can be changed to be declared as a static local function. To not access variables outside of its scope, parameters are defined to be passed when invoking this local function (code file `StaticLocalFunctionsSample/CollectionExtensions.cs`) :

```
public static IEnumerable<T> Where2<T>(
  this IEnumerable<T> source, Func<T, bool> pred)
{
  if (source == null) throw new ArgumentNullException(nameof(source));
  if (pred == null) throw new ArgumentNullException(nameof(pred));

  return Iterator(source, pred);

  static IEnumerable<T> Iterator(IEnumerable<T> source, Func<T, bool> pred)
  {
    foreach (T item in source)
    {
      if (pred(item))
        yield return item;
    }
  }
}
```

Readonly Members

Members of structs can now be declared with the `readonly` modifier. With C# 7 it is possible to declare the complete struct with a `readonly` modifier to make sure no field changes after creation. A `readonly` struct can only contain `readonly` fields and read-only properties. Using the `readonly` modifier with members of the struct is not that restrictive - the members that don't mutate state can be marked with the `readonly` modifier.

With the following code sample, the struct `SomeData` is defined. With this structure, `GetDataX` methods and the `DontChangeState` method are declared `readonly`. These methods do not change the state of the struct. These methods return fields and properties. Calling methods within `readonly` methods, these methods need to be declared `readonly` as well. Accessing properties from `readonly` methods you need to be aware that the `get` accessor is not automatically assumed to be read-only. The `get` accessor could change some state, so you need to modify it with the `readonly` modifier as well as shown with `Data3`. This is not necessary with auto implemented properties. Here the syntax is known to now change state with the `get` accessor - so with auto implemented properties the `readonly` accessor is not needed as shown with `Data4` and `Data5` properties (code file `CSharp8/ReadOnlyMembersSampe/Program.cs`):

```
public struct SomeData
{
  private readonly int _data1;
  private int _data2;
  public SomeData(int data1, int data2, int data3, int data4, int data5)
  {
    _data1 = data1;
    _data2 = data2;
    _data3 = data3;
    Data4 = data4;
    Data5 = data5;
  }

  private int _data3;
  public int Data3
  {
    readonly get => _data3;
    set => _data3 = value;
  }

  public int Data4 { get; set; }
  public int Data5 { get; }
  private void PrivateMethod() // not declared readonly
  {
    Console.WriteLine("PrivateMethod");
  }
```

```
  public readonly int GetData1() => _data1;
  public readonly int GetData2() => _data2;
  public readonly int GetData3() => Data3;
  public readonly int GetData4() => Data4;
  public readonly int GetData5() => Data5;

  public readonly void DontChangeState()
  {
    Console.WriteLine("DontChangeState");
    // PrivateMethod(); - cannot be invoked because this method is not readonly
  }
}
```

type="note"

Note

Be aware of the different way auto created properties and full properties are dealt with. The `get` accessor of auto created properties is automatically read-only while the get accessor of custom implemented properties needs to be declared with the `readonly` modifier.

Interpolated Verbatim Strings

A small but useful feature is an enhancement of interpolated verbatim strings. A verbatim string starts with the `@` token - to ignore the special characters in the string which is useful for example with regular expressions to not need to escape every special character. An interpolated string starts with the `$` token - to allow computed expressions within the string within curly braces. In case both is needed with a string, interpolated and verbatim strings, before C# 8 the order was important - you had to write `$@"…"` - the `$` token had to be before the `@` token. Now this limitation is removed, now you can also use `@$"…"`.

type="reference"

Reference

Interpolated and verbatim strings are discussed in Chapter 9, "Strings and Regular Expressions".

Default Interface Members

One of the most controversial C# 8 features is **default interface members**. With this feature, interfaces can include implementations. This is a major change from before and requires changes with the runtime.

Because of this, this feature is not available with the .NET Framework and requires at least .NET Core 3.0 and .NET Standard 2.1.

The major reason for this feature is versioning of interfaces. Before C# 8, any time you change an interface is a breaking change. Both callers and implementors need to be re-compiled, and implementors of an interface need to change the implementation to add new members of the interface. If you define interfaces only in your own projects with strong dependencies, this might not be an issue for you. However, if you create a library used by others, changing the interface means that any previously defined class that implements the interface no longer compiles.

type="note"

Note

Because of this versioning issue with interfaces, many implementations of .NET libraries derive from abstract classes instead of implementing interfaces. With new versions, abstract classes can be enhanced (if no abstract methods are added) without breaking derived classes.

Versioning of Interfaces

Let's get into an example. The interfaces `IPosition` and `IShape` are defined in the .NET Standard 2.1 library `SampleLib`:

```
public interface IPosition
{
  int X { get; set; }
  int Y { get; set; }
}

public interface IShape
{
  IPosition Position { get; set; }
}
```

These interfaces are implemented in the application `DefaultInterfaceMemberSample` (code file `DefaultInterfaceMembersSample/DefaultInterfaceMemberSample/Program.cs`):

```
public class Position : IPosition
{
  public int X { get; set; }
  public int Y { get; set; }
}

public class Shape : IShape
{
  private IPosition _position = new Position();
  public IPosition Position
  {
```

```
    get => _position;
    set => _position = value;
  }

  public override string ToString() => $"X: {Position.X}, Y: {Position.Y}";
}
```

Now, if one interface is changed with another member in a new version of the library, you can receive a compiler error. For example, if the library developer adds the `Move` method to `IShape`, rebuilding the application referencing the library results in the compiler error Shape does not implement interface member `'IShape.Move(IPosition)'`.

```
public interface IShape
{
  IPosition Position { get; set; }

  IPosition Move(IPosition newPosition);
}
```

Creating a default interface member instead – declaring the method and adding an implementation – the versioning issue can be solved. Adding a method with an implementation doesn't break versioning. The application using this library can be rebuilt using the new version of the library without changes (code file `DefaultInterfaceMembersSample/SampleLib/IShape.cs`):

```
public interface IShape
{
  IPosition Position { get; set; }

  public IPosition Move(IPosition newPosition)
  {
    Position.X = newPosition.X;
    Position.Y = newPosition.Y;
    return Position;
  }
}
```

Enhancing the calling app with new features, the `Move` method can be accessed by using the interface, e.g. by declaring the interface type as a method parameter, or by casting the object to the interface (code file `DefaultInterfaceMembersSample/DefaultInterfaceMemberSample/Program.cs`):

```
static void Move(IShape shape)
{
  shape.Move(new Position() { X = 99, Y = 99 });
}

static void Main()
{
  var shape = new Shape();
  shape.Position = new Position { X = 33, Y = 22 };
  Console.WriteLine(shape);
  (shape as IShape).Move(new Position { X = 44, Y = 33 });

  Move(shape);
```

```
  Console.WriteLine(shape);
}
```

As soon as you implement the default interface method in the derived class, this version is used – no matter if you use the variable as a type of the class or the interface. However, if you implement the method in the derived class you cannot call the default implementation from the interface anymore. Using the `base` keyword to invoke members from the interface is not possible – at least not with C# 8.

type="note"

Note

Invoking implementations of base interfaces was removed during the process creating default interface members from C# 8 but is planned for a future C# major version. See C# meeting notes https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-04-29.md#default-interface-implementations-and-base-calls

Trait

A **trait** is a concept which allows defining a set of methods to extend the functionality of a class. Some programming languages have a specific keyword for traits (e.g. PHP and Scala with the `trait` keyword, Ruby with `mixins`). C# now allows some functionality of traits with default interface members.

With C#, many extensions to different types are done with *extension methods*. Extension methods sometimes are hard to detect, as the namespace where the extension class is defined needs to be added. Similar functionality can be done using default interface members as show next.

type="note"

Note

Extension methods are covered in detail in Chapter 3, “Objects and Types”.

Many LINQ methods are implemented as extension methods to extend the `IEnumerable<T>` interface. Let’s have a look at using default interface members instead. Because we don’t have control on the `IEnumerable<T>` interface, the sample code defines the `ICustomEnumerable<T>` interface that just derives from `IEnumerable<T>`. The interface `ICustomEnumerable<T>` is created to define the `Where` trait to be used later instead of the `Where`

implementation that is implemented as an extension method in the `Enumerable` class. The `ICustomEnumerable<T>` interface defines an implementation of the `Where` method to filter elements based on the passed predicate (code file `CSharp8/DefaultInterfaceMemberSample/SampleLib/ICustomEnumerable.cs`):

```
public interface ICustomEnumerable<T> : IEnumerable<T>
{
  public IEnumerable<T> Where(Func<T, bool> pred)
  {
    foreach (T item in this)
    {
      if (pred(item))
      {
        yield return item;
      }
    }
  }
}
```

For using the interface with a collection, the class `CustomCollection<T>` is defined. This class derives from the `Collection<T>` base class (in the namespace `System.Collections.ObjectModel`) and implements the interface `ICustomEnumerable<T>`. Because `ICustomEnumerable<T>` doesn't define any additional members to `IEnumerable<T>` that need to be implemented, and `Collection<T>` already implements `IEnumerable<T>`, the implementation of this class can be kept simple (code file `CSharp8/DefaultInterfaceMembersSample/CustomCollection.cs`):

```
public class CustomCollection<T> : Collection<T>, ICustomEnumerable<T>
{
}
```

Next, the collection is used. The `GetCustomCollection` method returns a new instance passing some initial values. The method `CustomCollectionSample` demonstrates using the `Where` method both with the LINQ method syntax as well as the LINQ expression. Because there's a better match with the `ICustomEnumerable<T>` interface, the default interface members of the interface are used instead of the extension methods that extend the `IEnumerable<T>` interface (code file `CSharp8/DefaultInterfaceMembersSample/Program.cs`):

```
private static ICustomEnumerable<string> GetCustomCollection() =>
  new CustomCollection<string>
  { "James", "John", "Michael", "Lewis", "Jochen", "Juan" };
```

```
private static void CustomCollectionSample()
{
  var coll = GetCustomCollection();

  var subset = coll.Where(n => n.StartsWith("J"));
  foreach (var name in subset)
  {
    Console.WriteLine(name);
  }

  var subset2 = from n in coll
                where n.StartsWith("J")
                select n;
  foreach (var name in subset2)
  {
    Console.WriteLine(name);
  }
}
```

Multiple Inheritance

One way looking at default interface members is *multiple inheritance*. You cannot derive a class from multiple classes, but derivation from multiple interfaces is possible. Now as we can have implementations with interfaces, does this solve the problem of the `FlyingCarpet` class to derive both from `FlyingObject` and `Carpet`? The `CustomCollection<T>` class shown in the previous example derives from the class `Collection<T>`. There's also inheritance declaration from the interface `ICustomEnumerable<T>` to inherit the `Where` method. However, there's an important restriction. With default interface members you can declare methods with the interface, but you cannot keep any state. Declaring fields is not possible. It's possible to declare properties, but not fields associated with properties. This restriction doesn't allow using auto properties in interfaces.

What this restriction prohibits and what can be done around it is shown in a sample creating a base implementation for the interface `INotifyPropertyChanged`. This interface is used for change notification, e.g. when properties of an object change to automatically update user interfaces.

`type="reference"`

Reference

See Chapter 34, "Patterns with XAML Apps" on more information on change notification with `INotifyPropertyChanged`.

The interface `INotifyPropertyChanged` (in the namespace `System.ComponentModel`) defines the event `PropertyChanged` that needs to be implemented by the class deriving from this interface:

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}
```

Instead of creating a class implementing this interface, let's create the interface `IBindableBase` deriving from `INotifyPropertyChanged`. What's not possible with this interface is to implement the event from the base interface – this would keep state. This interface implements the method `SetProperty` which can be used by the class implementing this interface. However, the method `OnPropertyChanged` is declared, which needs to be implemented by the type implementing this interface. It's not possible to define a field for a delegate (using the event syntax) and fire the event directly from the interface (code file `DefaultInterfaceMembersSample/IBindableBase.cs`):

```
public interface IBindableBase : INotifyPropertyChanged
{
  void OnPropertyChanged(string propertyName);

  public virtual bool SetProperty<T>(
    ref T item, T value,
    [CallerMemberName] string propertyName = default!)
  {
    if (EqualityComparer<T>.Default.Equals(item, value)) return false;

    item = value;
    OnPropertyChanged(propertyName);
    return true;
  }
}
```

As entity types now do not need to derive from a common base class such as a `BindableBase`, other base classes can be used. Here, the `Entity` class is defined to specify an `Id` property (code file `DefaultInterfaceMembersSample/Entity.cs`):

```
public class Entity
{
  public int Id { get; set; }
}
```

The `Book` class now inherits the members from both the `Entity` class as well as the `IBindableBase` interface. Because `IBindableBase` derives from `INotifyPropertyChanged`, the Book class also needs to implement the members from this base interface – which is the `PropertyChanged` event. As defined by the `IBindableBase` interface, the `Book` class also implements the `OnPropertyChanged`

method. In this implementation, the `PropertyChanged` event is fired. With the implementation of the set accessor of the `Title` property, the `SetProperty` of the base interface is invoked, which in turn invokes the `OnPropertyChanged` method (code file `CSharp8/DefaultInterfaceMembersSample/Book.cs`):

```
public class Book : Entity, IBindableBase
{
  public event PropertyChangedEventHandler PropertyChanged = (s, e) => { };

  void IBindableBase.OnPropertyChanged(string propertyName) =>
    PropertyChanged.Invoke(this, new PropertyChangedEventArgs(propertyName));

  private string? _title;
  public string? Title
  {
    get => _title;
    set => (this as IBindableBase).SetProperty(ref _title, value);
  }
}
```

type="note"

Note

You need to be aware of the restriction with default interface members - you cannot define state with the interface. If you need state, use abstract base classes instead. In most scenarios, having an abstract base class implementing `INotifyPropertyChanged` is the easier scenario to work with – the implementation of the model types gets simpler.

Async Streams

One of the many great features of C# 8 is **async streams**. Before C# 8, you could use the `await` keyword only to get a single result – when the asynchronous method returns the result. This changed with C# 8. Using `await` it's now possible to get a stream of results. Result by result is returned asynchronously. This was made possible by defining asynchronous iterator interfaces, and updates to the `foreach` and the `yield` statements.

type="reference"

Reference

The `foreach` statement is covered in Chapter 2, “Core C#”. The `IEnumerator`, `IEnumerable` interfaces, and the `yield` statement are

covered in Chapter 7, “Arrays”. The task-based async pattern with the `await` keyword is covered in Chapter 15, “Asynchronous Programming.”

Asynchronous Interfaces

The new interfaces defined for async streams are `IAsyncEnumerable<T>`, `IAsyncEnumerator<T>`, and `IAsyncDisposable`.

The interface `IAsyncEnumerable<T>` looks like `IEnumerable<T>` with the exception that an `IAsyncEnumerator<T>` is returned instead of `IEnumerator<T>`:

```
public interface IAsyncEnumerable<[Nullable(2)] out T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator(
    CancellationToken cancellationToken = default);
}
```

The interface `IEnumerator<T>` defines a `Current` property and a `MoveNext` method. This is like `IAsyncEnumerator<T>`. With the interface `IAsyncEnumerator<T>`, the `MoveNextAsync` method returns a `ValueTask<bool>`. A value of `true` is returned if the next object is retrieved, `false` if it’s the end of the collection:

```
public interface IAsyncEnumerator<[Nullable(2)] out T> : IAsyncDisposable
{
  T Current { get; }

  ValueTask<bool> MoveNextAsync();
}
```

An enumerator also needs to be disposed – asynchronously. For this the interface `IAsyncDisposable` with the method `DisposeAsync` is defined:

```
public interface IAsyncDisposable
{
  ValueTask DisposeAsync();
}
```

type="note"

Note

Using the `await` statement is possible with any object offering the method `GetAwaiter`. Besides the `Task` class, the `ValueTask` struct implements the `GetAwaiter` method. The `ValueTask` struct is discussed in Chapter 21, “Tasks and Parallel Programming.”

Using yield with Async Streams

With the sample application for async streams a device is simulated that returns two random values after a random time. The data returned from the device is implemented in the class `SensorData`. This is a simple type with two read-only properties (code file `CSharp8/AsyncStreamsSample/SensorData.cs`):

```
public struct SensorData
{
  public SensorData(int value1, int value2)
    => (Value1, Value2) = (value1, value2);

  public int Value1 { get; }
  public int Value2 { get; }
}
```

The `ADevice` class utilizes the interface `IAsyncEnumerable<T>` in returned results by using the `yield` statement. `yield` was changed with C# 8 to not only implement synchronous iterators with the `IEnumerable<T>` and `IEnumerator<T>` interfaces, but also the `IAsyncEnumerable<T>` and `IAsyncEnumerator<T>` interfaces. In the method `GetSensorData1`, the `yield` statement is used to return sensor data with random values after a random time in an endless loop (code file `CSharp8/AsyncStreamsSample/ADevice.cs`):

```
public class ADevice
{
  public async IAsyncEnumerable<SensorData> GetSensorData1()
  {
    var r = new Random();
    while (true)
    {
      await Task.Delay(r.Next(300));
      yield return new SensorData(r.Next(100), r.Next(100));
    }
  }
}
```

type="reference"

Reference

The yield statement is explained in Chapter 12, “Language Integrated Query.”

type="note"

Note

The design team considered to not derive `IAsyncEnumerator<T>` from `IAsyncDisposable`. However, that design would complicate other parts, e.g. with pattern-based helpers that need to deal with different scenarios.

type="note"

Note

The interfaces `IAsyncDisposable` and `IAsyncEnumerator` return `ValueTask` with the methods `MoveNextAsync` and `DisposeAsync`. C# 7 was changed to allow awaits not only on the `Task` type, but instead with any type implementing the `GetAwaiter` method.

`ValueTask` is one of the types that can be used here. `ValueTask` is implemented as a *value type* instead of a *reference type* with the `Task`. With this, `ValueTask` doesn't have the overhead of an object in the managed heap. This can be useful iterating through a list where not every iteration really requires an asynchronous operation.

Using foreach

Next, let's iterate the asynchronous stream. The `foreach` statement has been extended with asynchronous functionality – making use of the asynchronous interfaces when the `await` keyword is used. With `await foreach`, one item is retrieved after the other – without blocking the calling thread (code file `CSharp8/AsyncStreamsSample/Program.cs`):

```
private static async Task UseEnumeratorAsync()
{
  var aDevice = new ADevice();

  await foreach (var x in aDevice.GetSensorData1())
  {
    Console.WriteLine($"{x.Value1} {x.Value2}");
  }
}
```

type="note"

Note

To run the application, you have to uncomment the invocation of `UseEnumeratorAsync` and the other methods using the enumerator in the `Main` method. `UseEnumeratorAsync` runs endlessly, so you need to stop the application run with CTRL+C.

Behind the scenes, the compiler translates the `async foreach` statement to a `while` loop, invocation of the `MoveNextAsync` method, and accessing the `Current` property. The enumerator is disposed with the async version of the using declaration. `await using` can be used with

the `IAsyncDisposable` interface (code file `CSharp8/AsyncStreamsSample/Program.cs`):

```
private static async Task WhileLoopAsync()
{
  var aDevice = new ADevice();

  IAsyncEnumerable<SensorData> en = aDevice.GetSensorData1();
  await using IAsyncEnumerator<SensorData> enumerator =
    en.GetAsyncEnumerator();
  while (await enumerator.MoveNextAsync())
  {
    var sensorData = enumerator.Current;
    Console.WriteLine($"{sensorData.Value1} {sensorData.Value2}");
  }
}
```

Cancellation

Creating a custom implementation of the `IAsyncEnumerator<T>` interface, to allow for cancellation, a `CancellationToken` parameter can be specified. The parameter is optional by supplying a default value and marked with the `EnumeratorCancellation` attribute. This way, either a cancellation token can be passed to the `GetSensorData2` method, or the cancellation token can be supplied in a different way invoking the `WithCancellation` method as shown next. The `CancellationToken` is passed along to the `Task.Delay` method – so this method is cancelled if the token tells so (code file `CSharp8/AsyncStreamsSample/SensorData.cs`):

```
public async IAsyncEnumerable<SensorData> GetSensorData2(
  [EnumeratorCancellation] CancellationToken cancellationToken = default)
{
  var r = new Random();
  while (true)
  {
    await Task.Delay(r.Next(500), cancellationToken);
    yield return new SensorData(r.Next(100), r.Next(100));
  }
}
```

type="reference"

Reference

The cancellation token is explained in detail in Chapter 21, “Tasks and Parallel Programming.”

Invoking the method `GetSensorData2`, in the code sample the cancellation token is not directly passed to this method (which is possible as well), but instead using the `WithCancellation` extension method.

The sample code sends a cancellation after 5 seconds. When cancellation happens, an `OperationCanceledException` is thrown which is caught in the calling code (code file `CSharp8/AsyncStreamsSample/Program.cs`):

```
private static async Task WithCancellationAsync()
{
  try
  {
    var cts = new CancellationTokenSource();
    cts.CancelAfter(5000);
    var aDevice = new ADevice();

    await foreach (var x in aDevice.GetSensorData2()
      .WithCancellation(cts.Token)
    {
      Console.WriteLine($"{x.Value1} {x.Value2}");
    }
  }
  catch (OperationCanceledException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

type="note"

Note

Since .NET Framework 4.0, .NET included the `IObserver<T>` and `IObservable<T>` interfaces. Reactive Extensions makes use of these interfaces. How does this compare to the new interfaces defined for *async streams*? With the `IObservable<T>` interface, the `Subscribe` method can be used to assign a subscriber and receive events. This explains the major difference between this model and the new async streams. Observables uses a **push**-based model, the sender is in control. The subscriber receives events when new items are available. With *async streams*, a **pull**-based model is used. The caller is in control when doing the next invocation of the `GetNextAsync` method and waits here to receive the result. Because of the async implementation, the caller continues only when the result is received, but the calling thread is not blocked and can continue other work.

Switch Expressions and Pattern Matching

The *switch statement* is already available since C# 1.0 and it's not a lot different to what was known from the C programming language (there's

a small difference). Now as we have Lambda expressions, it's time to simplify the switch functionality with the **switch expression**. This section also discusses enhancements with **pattern matching** that are very practical to use with the switch expression.

While the switch expression offers more readability compared to the switch statement (as soon as you are used to the lambda expressions), there's still a reason when to use the switch statement: the switch expression always returns a value. The switch statement can be used to invoke different actions without returning a value.

type="reference"

Reference

The switch statement is explained in Chapter 2, "Core C#", Pattern matching is covered in Chapter 13, "Functional Programming with C#."

Let's start with an existing sample from the book changing a switch statement to a switch expression. The class `BookTemplateSelector` is used in an UWP application to return a data template for the user interface depending on the value of a property. The `SelectTemplate` method is invoked for every item in a list to return either a template assigned to the `WroxBookTemplate`, or a `WileyBookTemplate`, or the `DefaultBookTemplate`. Here, the `switch` statement is used to decide on the `Publisher` property. Multiple `case` and `break` keywords are used to return the correct template:

```
public class BookTemplateSelector : DataTemplateSelector
{
  public DataTemplate DefaultBookTemplate { get; set; }
  public DataTemplate WroxBookTemplate { get; set; }
  public DataTemplate WileyBookTemplate { get; set; }

  public override DataTemplate SelectTemplate(object item,
    DependencyObject container)
  {
    var book = item as Book;
    if (book == null) return null;

    DataTemplate selectedTemplate = null;
    switch (book.Publisher)
    {
      case "Wrox Press":
        selectedTemplate = WroxBookTemplate;
        break;
      case "Wiley":
        selectedTemplate = WileyBookTemplate;
        break;
      default:
        selectedTemplate = DefaultBookTemplate;
        break;
    }
    return selectedTemplate;
  }
```

```
}
```

C# 7 already allows a small simplification on the method `SelectTemplate` with pattern matching. Instead of using the `as` operator to verify if the object item is a book, the `is` operator together with the type pattern checks for a `Book` type and assigns the item variable to the book variable – if the item is a `Book`:

```
public override DataTemplate SelectTemplate(object item,
  DependencyObject container)
{
  if (item is Book book)
  {
    DataTemplate selectedTemplate = null;
    switch (book.Publisher)
    {
      case "Wrox Press":
        selectedTemplate = WroxBookTemplate;
        break;
      case "Wiley":
        selectedTemplate = WileyBookTemplate;
        break;
      default:
        selectedTemplate = DefaultBookTemplate;
        break;
      }
      return selectedTemplate;
    }
    else
    {
      return null;
    }
  }
```

Another enhancement – still using the old `switch` statement – can be done with pattern matching and C# 7. Here, the switch is done on the `item` object itself, and patterns are used with the cases. Pattern matching is enhanced with the when clause. The first case checks for a `Book` with the type pattern and uses the `when` clause to verify if the `Publisher` property has the string value `"Wrox Press"`. With the third case, the variable of the book is not needed, thus assigned to a variable with the name `_`. Nowadays this variable name has a special meaning with some expressions to ignore it:

```
public override DataTemplate SelectTemplate(object item,
  DependencyObject container)
{
  DataTemplate selectedTemplate = null;
  switch (item)
  {
    case Book b when b.Publisher == "Wrox Press":
      selectedTemplate = WroxBookTemplate;
      break;
    case Book b when b.Publisher == "Wiley":
      selectedTemplate = WileyBookTemplate;
      break;
    case Book _:
```

```
      selectedTemplate = DefaultBookTemplate;
      break;
    default:
      selectedTemplate = null;
      break;
  }
  return selectedTemplate;
}
```

Next let's change this to the C# 8 switch expression with nullability enhancements as well. The complete class becomes a lot smaller. With the new switch expression, using the keyword and the expression are reversed. Instead of `switch (item)`, now `item switch` introduces the switch expression. The case and break keywords are no longer needed. Instead, the lambda operator `=>` has the case on the left side and the implementation on the right side. With the first case, a *type pattern* is used to check for the `Book` type. With pattern matching a new feature is to use **recursive patterns**. After one pattern matches, another is used to check for the next pattern. Using curly braces, the **property pattern** is introduced to check for the value of the `Publisher` property – to return the corresponding template. To make the switch expression exhaustive, to have cases for all scenarios, the _ pattern is used to fulfill every other case:

```
public class BookTemplateSelector : DataTemplateSelector
{
  public DataTemplate? DefaultBookTemplate { get; set; }
  public DataTemplate? WroxBookTemplate { get; set; }
  public DataTemplate? WileyBookTemplate { get; set; }

  public override DataTemplate? SelectTemplate(object item,
    DependencyObject container) =>
    item switch
    {
      Book { Publisher: "Wrox Press" } => WroxBookTemplate,
      Book { Publisher: "Wiley" } => WileyBookTemplate,
      Book _ => DefaultBookTemplate,
      _ => null
    };
}
```

type="note"

Note

The switch expression has some restrictions compared to the switch statement. You might still need to use the switch statement in some cases. The switch expression always needs to return a value. With the switch statement you can do an action in every case and not return a value. This is not possible with the switch expression. Another restriction with the switch expression is that in the implementation of a case only one expression can be used. You cannot use curly braces to write multiple lines of code. Of course, you can invoke a method that returns a value.

> Here, local functions come in handy when you need the implementation only with the switch.

State changes with the switch expression

Let's get into another sample to see the features of the switch expression – with states of a traffic light. First, states are defined with the `enum` keyword - `Red`, `Yellow`, and `Green`:

```
public enum LightState
{
  Undefined,
  Red,
  Yellow,
  Green,
};
```

This is a very simple scenario. Changing from one light to the other, just the cases can be used to return a new state based on the current state. If the light is red, yellow is returned. If the light is yellow, green is returned:

```
public LightState GetNextLight1(LightState currentLight) =>
  currentLight switch
  {
    LightState.Red => LightState.Yellow,
    LightState.Yellow => LightState.Green,
    LightState.Green => LightState.Red,
    _ => LightState.Undefined
  };
```

Adding a static using to open the `LightState`, the implementation can even be simplified:

```
using static SwitchSample.LightState;
```

Now just the values of the enum can be used - which can enhance readability (code file `CSharp8/SwitchStateSample/TrafficLightSwitcher.cs`):

```
public LightState GetNextLight1(LightState currentLight) =>
  currentLight switch
  {
    Red => Yellow,
    Yellow => Green,
    Green => Red,
    _ => Undefined
  };
```

The `TrafficLightRunner` repeats invocations of the `GetNextLight1` method after a delay. The light switches from one state to the next (code file `CSharp8/SwitchStateSample/TrafficLightRunner.cs`):

```
public class TrafficLightRunner
{
  private readonly TrafficLightSwitcher _switcher = new TrafficLightSwitcher();
```

```
  public async Task SimpleLigthAsync()
  {
    LightState current = LightState.Red;
    while (true)
    {
      current = _switcher.GetNextLight1(current);
      Console.WriteLine($"new light: {current}");
      await Task.Delay(2000);
    }
  }
  //...
}
```

type="note"

Note

Starting the application, you need to pass command-line arguments like `--mode=Simple` to run it with the different options. This application makes use of the NuGet package `System.CommandLine.Experimental` to read and behave differently based on the passed command line.

Using tuples with switch expressions

For quite simple scenarios, the switch expression can be used to change from one state to the next. However, the traffic light is not really that simple. The light switches from yellow to either red or green, depending what the previous state was: read - yellow - green - yellow - red.

The traffic light is also more complex. Depending on the country of the traffic light, the green light can flash multiple times before switching to yellow. There's also a flashing yellow state, so the `LightState` enum type is changed to cover the different options (code file `CSharp8/SwitchStateSample/LightState.cs`):

```
public enum LightState
{
  Undefined,
  Red,
  Yellow,
  FlashingGreen,
  Green,
  FlashingYellow
};
```

With a new version of the light switching method, `GetNextLight2`, this all is paid attention to. To use a combination of the current and previous light what's needed to decide the next switch after yellow, a tuple comes in handy.

type="reference"

> **Reference**
>
> Tuples are discussed in detail in Chapter 13, "Functional Programming with C#."

The method `GetNextLight2` receives the current and previous light with the parameters and is declared to return a tuple containing the new current and previous values. The two parameters are combined to a tuple using parentheses, and the switch expression works on the tuple. With the cases of the switch expression, the **tuple pattern** is used. If the state received has a current value of yellow and a previous value of read, the next current value is green. With a current value of yellow and a previous value of flashing green, the next current value is red. With the other cases the previous value is ignored using the **discard pattern** `_`. If the current value is red, the next current value is yellow, no matter what the previous state was (code file `CSharp8/SwitchStateSample/TrafficLightSwitcher.cs`):

```
public (LightState Current, LightState Previous) GetNextLight2(
  LightState currentLight, LightState previousLight) =>
  (currentLight, previousLight) switch
  {
    (FlashingYellow, _)     => (Red, currentLight),
    (Red, _)                => (Yellow, currentLight),
    (Yellow, Red)           => (Green, currentLight),
    (Green, _)              => (FlashingGreen, currentLight),
    (FlashingGreen, _)      => (Yellow, currentLight),
    (Yellow, FlashingGreen) => (Red, currentLight),
    _                       => (FlashingYellow, currentLight)
  };
```

The invocation of `GetNextLight2` now needs to pass the current and the previous state, and receives a tuple from the method as shown in the method `UseTuplesAsync` (code file `CSharp8/SwitchStateSample/TrafficLightRunner.cs`):

```
public async Task UseTuplesAsync()
{
  LightState current = LightState.FlashingYellow;
  LightState previous = LightState.Undefined;
  while (true)
  {
    (current, previous) = _switcher.GetNextLight2(current, previous);
    Console.WriteLine($"new light: {current}, previous: {previous}");
    await Task.Delay(2000);
  }
}
```

Using expressions to create tuples

To let the traffic light keep the state of the flashing green 3 times, an additional counter needs to be introduced. Instead of a tuple with two

values, a tuple with tree values can be used. The third value is now the count which is used with the green flashing light. If the current light is flashing green, and the count has a value of two, the light switches to yellow. With a flashing green and all other count values, the `currentCount` variable is incremented, and the new value returned (code file `CSharp8/SwitchStateSample/TrafficLightSwitcher.cs`):

```
public (LightState Current, LightState Previous, int count)
  GetNextLight3(
  LightState currentLight, LightState previousLight, int currentCount = 0) =>
  (currentLight, previousLight, currentCount) switch
  {
    (FlashingYellow, _, _)     => (Red, currentLight, 0),
    (Red, _, _)                => (Yellow, currentLight, 0),
    (Yellow, Red, _)           => (Green, currentLight, 0),
    (Green, _, _)              => (FlashingGreen, currentLight, 0),
    (FlashingGreen, _, 2)      => (Yellow, currentLight, 0),
    (FlashingGreen, _, _)      => (FlashingGreen, currentLight,
      ++currentCount),
    (Yellow, FlashingGreen, _) => (Red, currentLight, 0),
    _                          => (FlashingYellow, currentLight, 0)
  };
```

Calling the `GetNextLight3` method, the count is passed with the third argument, and received with the tuple (code file `CSharp8/SwitchStateSample/TrafficLightRunner.cs`):

```
public async Task UseTuplesWithCountAsync()
{
  LightState current = LightState.FlashingYellow;
  LightState previous = LightState.Undefined;
  int count = 0;
  while (true)
  {
    (current, previous, count) =
      _switcher.GetNextLight3(current, previous, count);
    Console.WriteLine($"new light: {current}, previous: {previous}, " +
      $"count: {count}");
    await Task.Delay(2000);
  }
}
```

Switch expressions with the property pattern

So far, the traffic light runner changes the light every two seconds. Of course, this is not a real traffic light scenario. Depending on the light, different timings should be used. Another value could be put into the tuple, but at some point, instead of using tuples, custom types help with readability. Now is the time to create the `LightStatus` struct defining `Current`, `Previous`, `FlashCount`, and `Milliseconds` properties (code file `CSharp8/SwitchStateSample/LightStatus.cs`):

```
public readonly struct LightStatus
```

```
{
  public LightStatus(LightState current, LightState previous,
    int seconds, int blinkCount) =>
    (Current, Previous, Milliseconds, FlashCount) =
      (current, previous, seconds, blinkCount);

  public LightStatus(LightState current, LightState previous, int seconds)
    : this(current, previous, seconds, 0) { }

  public LightStatus(LightState current, LightState previous)
    : this(current, previous, 3) { }

  public LightState Current { get; }
  public LightState Previous { get; }
  public int FlashCount { get; }
  public int Milliseconds { get; }
}
```

Instead of using the tuple pattern, now the property pattern is used to switch on specific values based on the properties. A new instance of the `LightStatus` is returned with the next value (code file `CSharp8/SwitchStateSample/TrafficLightSwitcher.cs`):

```
public LightStatus GetNextLight4(LightStatus lightStatus) =>
  lightStatus switch
  {
    { Current: FlashingYellow } => new LightStatus(Red, FlashingYellow, 5000),
    { Current: Red } => new LightStatus(Yellow, lightStatus.Current, 3000),
    { Current: Yellow, Previous: Red } =>
      new LightStatus(Green, lightStatus.Current, 5000),
    { Current: Green } =>
      new LightStatus(FlashingGreen, lightStatus.Current, 1000),
    { Current: FlashingGreen, FlashCount: 2 } =>
      new LightStatus(Yellow, lightStatus.Current, 2000),
    { Current: FlashingGreen } =>
      new LightStatus(FlashingGreen, lightStatus.Current, 1000,
        lightStatus.FlashCount + 1),
    { Current: Yellow, Previous: FlashingGreen } =>
      new LightStatus(Red, lightStatus.Current, 5000),
    _ => new LightStatus(FlashingYellow, lightStatus.Current, 1000)
  };
```

With the next version on using the traffic light switcher, a new `LightStatus` is created, and the delay is now based on the number of milliseconds returned (code file `CSharp8/SwitchStateSample/TrafficLightRunner.cs`):

```
public async Task UseCustomTypeAsync()
{
  var lightStatus = new LightStatus();
  while (true)
  {
    lightStatus = _switcher.GetNextLight4(lightStatus);
    Console.WriteLine($"new light: {lightStatus.Current}, " +
      $"previous: {lightStatus.Previous}, " +
      $"count: {lightStatus.FlashCount}, " +
      $"time: {lightStatus.Milliseconds}");
    await Task.Delay(lightStatus.Milliseconds);
  }
}
```

With the flow of the switch expression sample you've not only seen different uses of the new switch expression with tuples and pattern matching, but also a modern functional approach using immutable types. The `LightStatus` type is declared `readonly` and a new instance is returned with every iteration. Before using the `LightStatus` type, tuples have been used, similar to the `LightStatus` with immutability in mind.

Indices and Ranges

Indices and ranges make it easier to access a range of data from strings, arrays, and collections. An interesting aspect here is that .NET Core 2.1 added the `Span` type. With the `Span` type, memory (no matter if it's on the heap or the stack) can be directly accessed, and it's easy to directly access a split of the memory. Since the `Span` type has been released, many existing APIs have new overloads where it's no longer necessary to deal with memory arrays and the length of the memory to access, but instead the `Span` can be used that knows itself about the length. While the `Span` makes it easier to use these APIs, indices and ranges now offer direct support from C#.

`type="reference"`

Reference

The `Span` type is explained in detail in Chapter 17, "Managed and Unmanaged Memory."

Requirements

Some C# features require specific types from the framework. For example, interpolated strings are based on the `FormattableString` class. The `foreach` statement makes use of `IEnumerable`, and `IEnumerator` interfaces. The `using` statement and declaration use the `IDisposable` interface. These are just a few examples, and there are a lot more. The new C# 8 feature with indexes and ranges uses the `Index` and `Range` structs.

From Byte Array to Span to Ranges

To see the advantages of his new C# feature, let's start a small application. With this application, a byte array buffer should be filled. The

first six bytes of this buffer should be filled with a preamble – 6 times the hex value 42, followed by the content of a file. The file is stored in the UTF8 format, and the 3 bytes for the BOM (byte order mark) should be removed. The 8 last bytes of the buffer should not be filled, so the remaining content of the file should just be ignored.

Using byte arrays and offsets

The first version of the sample makes use of byte array and offsets in the method `BufferWithOffsetAndCount`. One overload of the `Read` method from the `StreamReader` class allows passing a byte array as the first parameter, the offset where the write into the byte array should start, and the number of bytes that should be read. The offset is here set to `3` to write the first characters (after the 3-byte BOM) at index 6. The maximum count of the bytes to read is calculated based on the length of the buffer reduced by 8 and not to forget the offset. After the data is read, the first 6 bytes in the buffer are initialized to the hex value `42` which also overwrites the BOM (code file `CSharp8/RangesSample/Program.cs`):

```
private static void BufferWithOffsetAndCount()
{
  Console.WriteLine("buffer with offset and count");

  byte[] buffer = new byte[64];
  using Stream stream = File.Open("QuickFox.txt", FileMode.Open);

  int offset = 3;
  int read = stream.Read(buffer, offset, count: buffer.Length - 8 - offset);
  Console.WriteLine($"read {read} bytes");

  byte init = 0x_42;
  for (int i = 0; i < 6; i++)
  {
    buffer[i] = init;
  }

  string s = Encoding.UTF8.GetString(buffer);
  Console.WriteLine(s);
  Console.WriteLine();
}
```

Using Span

The `Span` type allows some simplifications. Using the extension method `AsSpan` with the byte array, a `Span` is returned that covers the complete byte array. To read the file, another `Span` is created using the `Slice` method. `Slice` returns a slice into the span passing the start value and the length. The `Read` method of the `StreamReader` offers an overload with a Span parameter. This way you don't need to pass a start

position and a length because the `Span` knows about this itself. The length still needs to be calculated on creating the slice. Filling the preamble with the initial value, another `Span` is created. The `Fill` method is used to fill the complete slice (code file `CSharp8/RangesSample/Program.cs`):

```
private static void BufferWithSpan()
{
  Console.WriteLine("buffer with span");

  byte[] buffer = new byte[64];
  var bufferSpan = buffer.AsSpan();
  using Stream stream = File.Open("QuickFox.txt", FileMode.Open);

  int offset = 3;
  var spanForFile = bufferSpan.Slice(
    start: offset, length: bufferSpan.Length - 8 - offset);
  int read = stream.Read(spanForFile);
  Console.WriteLine($"read {read} bytes");

  byte init = 0x_42;
  bufferSpan.Slice(0, 6).Fill(init);

  string s = Encoding.UTF8.GetString(buffer);
  Console.WriteLine(s);
  Console.WriteLine();
}
```

Using Ranges with Span

With the new C# feature, another simplification can be done with this scenario. Instead of using the `Slice` method of the `Span` type, a range can be used. The second range is used to initialize the first six elements. A range is defined using brackets containing .. (two dots). The first value in the range specifies the start of the range. `0` defines the first element. The first element starts with 0 as we are used to in C# using an indexer. The second value in the range specifies the element after the last element accessed, so the first six elements are accessed with the range `0..6`. The first range specified makes use of the **hat operator** `^`. Using the *hat operator*, you can access elements starting from the last one. `^0` specifies the element after the last element, `^1` one element before that – the last element (code file `CSharp8/RangesSample/Program.cs`):

```
private static void BufferWithSpanAndRanges()
{
  Console.WriteLine("buffer with span");

  byte[] buffer = new byte[64];
  var bufferSpan = buffer.AsSpan();
  using Stream stream = File.Open("QuickFox.txt", FileMode.Open);

  int offset = 3;
  var spanForFile = bufferSpan[3..^8];
```

```
  int read = stream.Read(spanForFile);
  Console.WriteLine($"read {read} bytes");

  byte init = 0x_42;
  bufferSpan[0..6].Fill(init);

  string s = Encoding.UTF8.GetString(buffer);
  Console.WriteLine(s);
  Console.WriteLine();
}
```

Index

As previously mentioned, to support the new index syntax, the `Index` struct is defined. This type defines `Value` and `IsFromEnd` properties. If `IsFromEnd` returns true, the value of the index is used to access the collection from the end. The `GetOffset` method returns the offset of the indexed element from the begin of the collection.

In the next code sample, the index is used to access a simple array. `ix1` is defined to access the first element, `ix2` to access the last element with the help of the hat operator, and `ix3` is created with the constructor of the `Index` struct to access the third-last element (code file `CSharp8/RangesSample/Program.cs`):

```
private static void IndexSample()
{
  int[] data = { 1, 2, 3, 4, 5, 6 };
  Index ix1 = 0;
  Index ix2 = ^1;
  Index ix3 = new Index(3, fromEnd: true);
  ShowIndices(ix1, ix2, ix3);

  void ShowIndices(params Index[] indices)
  {
    foreach (var ix in indices)
    {
      Console.WriteLine($"value: {ix.Value}, " +
                        $"is from end: {ix.IsFromEnd}, " +
                        $"offset: {ix.GetOffset(length)}");
      Console.WriteLine($"value of array element: {data[ix]}");
    }
  }
}
```

Running the application, you'll see output as shown here:

```
value: 0, is from end: False, offset: 0
value of the array element: 1
value: 1, is from end: True, offset: 5
value of the array element: 6
value: 3, is from end: True, offset: 3
value of the array element: 4
```

Ranges

To see the features of ranges, the method `RangesSample` is defined. Six ranges are specified to access elements of an int array. `r1` references the first up to the second element (remember the end specified defines the element after). `r2` uses the hat operator to reference elements. The first element that's referenced from this range is the fourth-last element, and the range goes up to the second-last element. `r3` defines a range that goes up to the last element, starting from the fourth element. `r4` starts with the first element up to the fourth. `r5` references the complete range. Besides using the range-syntax you can also use static members of the Range struct to create a range. This is shown with the `StartAt` method used to set the `r6` variable. Besides `StartAt` you can also use the `EndAt` method and the `All` property. With the `ShowRanges` method, information about the range including the elements referenced by the range are shown. Instance members of the `Range` type are the `Start` and `End` properties, and the `GetOffsetAndLength` method that returns a tuple containing the offset and the length (code file `CSharp8/RangesSample/Program.cs`):

```
private static void RangesSample()
{
  int[] data = { 1, 2, 3, 4, 5, 6 };
  Range r1 = 0..2;
  Range r2 = ^4..^2;
  Range r3 = 3..;
  Range r4 = ..4;
  Range r5 = ..;
  Range r6 = Range.StartAt(4);

  ShowRanges(r1, r2, r3, r4, r5, r6);
  Console.WriteLine();

  void ShowRanges(params Range[] ranges)
  {
    foreach (var r in ranges)
    {
      (var offset, var length) = r.GetOffsetAndLength(data.Length);
      Console.WriteLine($"range start: {r.Start}, " +
                        $"end: {r.End}, offset: {offset}, " +
                        $"length: {length}, " +
                        $"content: {string.Join(' ', data[r])}");
    }
  }
}
```

Running the application, you can see output as shown.

```
range start: 0, end: 2, offset: 0, length: 2, content: 1 2
range start: ^4, end: ^2, offset: 2, length: 2, content: 3 4
range start: 3, end: ^0, offset: 3, length: 3, content: 4 5 6
range start: 0, end: 4, offset: 0, length: 4, content: 1 2 3 4
range start: 0, end: ^0, offset: 0, length: 6, content: 1 2 3 4 5 6
```

```
range start: 4, end: ^0, offset: 4, length: 2, content: 5 6
```

Range and Index with String Arrays

So far byte arrays have been used in the samples. You can use ranges and indices with any array types. In the following code snippet, an index is used to access one string in the string array, and a range is used in the foreach loop to access a range of strings (code file `CSharp8/RangesSample/Program.cs`):

```
private static void RangeAndIndexWithStringArray()
{
  string[] names = { "James", "Niki", "Jochen", "Juan", "Michael",
                     "Sebastian", "Nino", "Lewis" };

  string lewis = names[^1];  // uses an index to access Lewis
  Console.WriteLine(lewis);
  foreach (var name in names[2..^2]) // uses a range
  {
    Console.WriteLine(name);
  }
}
```

Range and Index with Strings

A string itself is a collection of characters - so you can use ranges and indices directly with strings as well. The `Substring` method of the `String` class is no longer required. The following code snippet defines the string fox1 and uses several syntax variants to use an index and ranges to access characters of the string. To pass both the name of the variable and the range variable in an array list, the local function `ShowStrings` uses an array of tuples (code file `CSharp8/RangesSample/Program.cs`):

```
private static void RangeAndIndexWithStrings()
{
  string fox1 = "the quick brown fox jumped over the lazy dogs";
  string quick = fox1[4..9];
  string dog = fox1[^4..^1];
  string brownfoxjumped = fox1[10..];
  string thequick = fox1[..9];
  string fox2 = fox1[..];

  Console.WriteLine($"character accessed with index: {fox1[^2]}");

  ShowStrings(
    (nameof(fox1), fox1),
    (nameof(quick), quick),
    (nameof(dog), dog),
    (nameof(brownfoxjumped), brownfoxjumped),
    (nameof(thequick), thequick),
    (nameof(fox2), fox2));
```

```
  static void ShowStrings(params (string Name, string Value)[] vals)
  {
    Console.WriteLine(nameof(RangeAndIndexWithStrings));
    foreach (var s in vals)
    {
      Console.WriteLine($"{s.Name}, {s.Value}");
    }
    Console.WriteLine();
  }
}
```

Range and Index with Custom Collections

For using ranges and indices with custom collections, one way is to offer methods, properties, and indexers with `Index` and `Range` parameters. However, there's also an easier option. These new operators are pattern-based. To offer the index syntax with custom collections is that the collection needs to be countable (by offering a `Length` or `Count` property), and an indexer with int parameter needs to be available. To use the range syntax, a method named `Slice` with int start and length parameters needs to be available. The return type of the Slice method is not relevant for a match to this method.

The sample collection is kept simple by using an int array internally which is filled with numbers from 0 to 99. The `Length` property and the indexer are implemented to support `Index`. The `Slice` method accepts start and length parameters to support `Range` (code file `CSharp8/RangesSample/MyCollection.cs`):

```
public class MyCollection
{
  private int[] _array = Enumerable.Range(0, 100).ToArray();

  public int Length => _array.Length;

  public int this[int index]
  {
    get => _array[index];
    set => _array[index] = value;
  }

  public int[] Slice(int start, int length)
  {
    var slice = new int[length];
    Array.Copy(_array, start, slice, 0, length);
    return slice;
  }
}
```

Reference

Implementing of indexers is explained in Chapter 6, “Operators and Casts.”

The next code snippet shows how to use the custom collection. The index and range operators can be used as expected. One element is accessed passing an index with the hat operator, and a range of the collection is accessed using the range operator (code file `CSharp8/RangesSample/Program.cs`):

```
var coll1 = new MyCollection();
int element = coll1[^3];
Console.WriteLine(element);

var range = coll1[11..15];
foreach (var item in range)
{
  Console.WriteLine(item);
}
```

What about using existing collection classes such as `List<T>` with index and range operators? The `List<T>` class already offers a `Count` property and an indexer, so you can use the index operator with this type. However, this class doesn’t have implementations for ranges. It’s not possible to create a Slice extension method with two int parameters. Extension methods don’t match the pattern-based mapping for the range operator (at least with C# 8 and 9). As a workaround you can create an extension method where a `Range` parameter is used. The following code snippet shows an extension method named `Slice` to extend every type implementing `IList<T>` with a `Range` parameter. After getting the offset and length, the `Skip` and `Take` methods are used to return a subset (code file `CSharp8/RangesSample/ListExtensions.cs`):

```
public static class ListExtensions
{
  public static IEnumerable<T> Slice<T>(this IList<T> list, Range range)
  {
    (var offset, var length) = range.GetOffsetAndLength(list.Count);
    return list.Skip(offset).Take(length).ToList();
  }
}
```

With this extension method in place, using the `List<T>` class it’s not only possible to use the index operator, but also to invoke the `Slice` method and pass a range (code file `CSharp8/RangesSample/Program.cs`):

```
List<int> list1 = new List<int>() { 1, 2, 3, 4, 5, 6 };
int item1 = list1[^2];
Console.WriteLine(item1);

var list2 = list1.Slice(2..^1);
foreach (var item in list2)
{
```

```
  Console.WriteLine(item);
}
```

Summary

This chapter examined all the new features of C# 8. Productivity features such as switch expressions, enhancements with pattern matching, using declarations, and async streams allow to reduce the code you need to write. Nullable reference types will reduce errors. The most common exception with .NET is the `NullReferenceException` which should go away by a large amount when nullable reference types are used everywhere. Default interface members allow for non-breaking changes with interfaces. Indices and ranges can simplify the code and thus also enhance productivity.