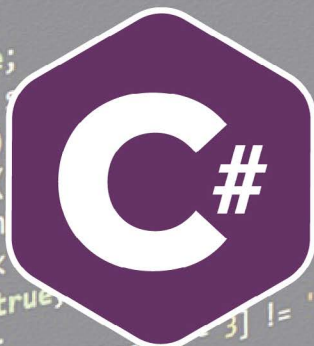


COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY THE AWESOME

NOW INCLUDES
CHAPTERS ON
.NET Core 3.0
& C# 8.0

BOOK ON



AND

.NET

DAMIR ARH

The Absolutely Awesome
Book on

C# and .NET

(Covers C# v6, v7, and .NET Core)

By Damir Arh

The Absolutely Awesome Book on C# and .NET

Copyright (c) DotNetCurry on behalf of A2Z Knowledge Visuals Pvt Ltd.

ALL RIGHTS RESERVED. No part of this eBook or related content may be reproduced, duplicated, given away, transmitted or resold in any form or by any electronic or mechanical means (electronic, photocopying, recording, or otherwise), including information storage and retrieval systems without written prior permission from DotNetCurry.com.

Please note that much of this publication is based on the practical experience of the author. Although the author has made every reasonable attempt to achieve complete accuracy of the content in this eBook and related content, he assumes no responsibility for errors or omissions or completeness. Also, you should use this information as you see fit, and at your own risk. Your particular situation may not be exactly suited to the text and examples illustrated here; in fact, it's likely that they won't be the same, and you should adjust your use of the information and recommendations, accordingly.

Any trademarks, service marks, product names or named features are assumed to be the property of their respective owners, and are used only for reference. There is no implied endorsement if we use one of these terms.

Bulk Sales

DotNetCurry.com offers bulk discount on this EBook for libraries and companies.

For more information, please contact A2Z Knowledge Visuals Pvt. Ltd at

ebooks@a2zknowledgevisuals.com

Cover Image By *Minal Agarwal* (minalagarwal@a2zknowledgevisuals.com).

 | Knowledge Visuals



Published in 2019 by A2Z Knowledge Visuals Pvt Ltd.

Cover created by **Minal Agarwal**

EBook created by **Minal Agarwal**

EBook Conceptualization by **Suprotim Agarwal**

Editing and Proofreading by **Suprotim Agarwal**

EBook Production and Distribution by *DotNetCurry.com* (A subsidiary of
A2Z Knowledge Visuals Pvt Ltd)

Written by Damir Arh and Reviewed by Yacoub Massad

About The Author



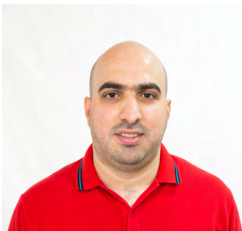
Damir Arh has many years of experience with software development and maintenance; from complex enterprise software projects, to modern consumer oriented mobile applications. Although he has worked with a wide spectrum of different languages, his favourite language remains C#. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his

knowledge by speaking at local user groups and conferences, and by blogging, and writing articles.

He has received the prestigious [Microsoft MVP award](#) for developer technologies for 7 times in a row.

In his spare time, he's always on the move: hiking, geocaching, running, and sport climbing. You can follow him on twitter [@damirarh](#) or read his tutorials at www.dotnetcurry.com/author/damir-arh

About The Reviewer



Yacoub Massad is a software developer and architect who works primarily on Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software.

You can follow him on twitter [@yacoubmassad](#) and view his blog posts at criticalsoftwareblog.com.

Preface

I've been using C# regularly ever since the first beta version of .NET framework 1.0, released way back in 2001. At first, I started as a developer, and later started sharing my learnings and experience via blogging. As I gained more knowledge, I started speaking at local user groups and conferences. These activities motivated me to deep dive into the subject, so that I could explain them better and answer more questions I was asked.

Eventually I was recognized as a Microsoft MVP (Most Valuable Professional) for my community contributions. This opened more opportunities for me, including writing articles for the [DotNetCurry \(DNC\) Magazine](#) and now writing this book.

My experience so far reflects in how I have approached the book. I tend to explain the concepts in my own words using examples of dos and don'ts based on my own work and speaking experience. There were topics in the book that required me to do additional research and learn something new in order to explain them in a way I was satisfied with. My ultimate goal for every explanation was that the information should be comprehensible to readers.

Target audience

The book assumes familiarity with basic programming concepts. Prior experience with C# and .NET framework or .NET Core is recommended. The book will be most beneficial to readers who are already confident with the basics of C# and .NET development, and would like to deepen their knowledge and understanding. The book structure also lends itself well to experienced developers who want to catch up with the latest features or refresh their knowledge about a specific feature or topic.

What you will learn

.NET and C# have been around for a very long time, but their constant growth means there's always more to learn. To keep up with the ever-changing .NET landscape, sometimes you need succinct and easy to follow information, with just enough detail to speedily grasp the concept, and keep moving ahead. This eBook will cover:

- The difference between .NET framework, .NET Core and .NET Standard, and the current state of all three.
- The role of C#, the Common Language Runtime (CLR) and the Base Class Library (BCL) in C#/.NET development.

- Important details about the C# type system and the language support for object-oriented programming (OOP).
- An overview of built-in collection classes and how they are closely related to generics and the Language INtegrated Queries (LINQ).
- Everything you need to know about parallel and asynchronous programming in C# (with emphasis on the `async` and `await` keywords).
- Key recent language developments in C# 6.0, C# 7.0 and C# 8.0.

Software requirements

Most code in the book will work with .NET Core 2.2 on any supported platform. You can find installation instructions on [the download page](#). Exact prerequisites for each platform are listed in the documentation ([Windows](#), [Linux](#) or [macOS](#)). The only exceptions are:

- Windows-specific features (Windows Forms, Windows Presentation Foundation (WPF) and ASP.NET) require the latest version of .NET framework running on Windows.
- C# 8 topics require .NET Core or later.

Sample code

All code from the book is organized in ready-to-run projects and grouped by chapters. Whenever possible, these are cross-platform .NET Core projects which run on every supported platform from a code editor like Visual Studio 2017 or Visual Studio Code, and even from command line. For exceptions, requirements are explained in the README.md file of each chapter folder.

You are encouraged to run the code, modify it and further experiment with it.

Acknowledgements

First and foremost, I must thank [Suprotim Agarwal](#) to make this book possible. He originally contacted me with the idea for the book, helped me finalize the table of contents and most importantly, he took care of the publishing process.

[Yacoub Massad](#) double checked all the facts stated in the book and made sure that the book is as accurate and comprehensible as possible. More than once, he encouraged me to rewrite a paragraph or rework an example which I had already considered done. I wasn't always happy with it at the time, but in hindsight, I'm glad he did.

There are too many people I have interacted with in the course of my career to list individually, who helped me recognize what developers struggle with. Thanks to them, many explanations and examples in this book are easier to comprehend.

Finally, I must thank Anders Hejlsberg, Mads Torgersen and everyone else who worked on the C# compiler at some point in time. If it weren't for them, there wouldn't be C# and I couldn't have written this book.

To each and every one, once again, thank you very much!

Contents

SECTION I

.NET AND THE COMMON LANGUAGE RUNTIME (CLR)

- Q1.** Which platforms can I develop applications for using the .NET framework? 17
- Q2.** How is .NET Core different from the .NET framework? 21
- Q3.** What is .NET Standard and why does it matter to me? 25
- Q4.** How can I create a .NET Standard library and use it from different runtimes? 30
- Q5.** What is the Roslyn compiler and how can I use it? 37
- Q6.** How did the .NET Framework evolve since version 1.0? 41
- Q7.** How does the version of .NET and C# affect compatibility? 47
- Q8.** What is the best way to consume third party libraries? 53
- Q9.** What are strong-named assemblies and how do I create them?58
- Q10.** What should I know about Garbage Collection in .NET? 64
- Q11.** How are value and reference types different? 69
- Q12.** What is the difference between classes and structs? 73

SECTION II

THE TYPE SYSTEM

Q13. What is boxing and unboxing?	79
Q14. Why are nullable types applicable only to value types?	85
Q15. How are enumeration types supported in C#?	88
Q16. How can multi-dimensional data be modelled using arrays? ...	93
Q17. What are some of the less commonly used C# operators?	98
Q18. How can method parameters be passed by reference?	103
Q19. When does using anonymous types make sense?	108
Q20. What does it mean that C# is a type safe language?	113
Q21. How can dynamic binding be implemented in C#?	118

SECTION III

CLASSES AND INHERITANCE

Q22. What is polymorphism and how to implement it in C#?	127
Q23. What is the meaning of individual accessibility levels?	131
Q24. What are the advantages and disadvantages of abstract classes over interfaces?	136
Q25. When should one explicitly implement an interface member?	140
Q26. What does the "new" modifier on a method mean?	144
Q27. How to correctly override the Equals method?	148

-

Q28. When and how to overload operators?	155
Q29. How do delegates and events differ?	161
Q30. What's special about the IDisposable interface?	168
Q31. How do Finalizers work?	174
Q32. What is the purpose of Static class members?	179
Q33. When are Extension Methods useful?	185
Q34. What are auto-implemented properties and why are they better than public fields?	189
Q35. What are Expression-bodied members?	193

SECTION IV

STRING MANIPULATION

Q36. When can string interpolation be used instead of string formatting?	199
Q37. How do locale settings affect string manipulation?	206
Q38. When can string manipulation performance suffer and how to avoid it?	210

SECTION V

GENERIC TYPES AND COLLECTIONS

Q39. How can Generics improve code?	215
Q40. How to implement a Generic method or class?	219
Q41. What are Generic type constraints?	225

Q42. What is Covariance and Contravariance?	231
Q43. How does the compiler handle the IEnumerable<T> interface?.....	236
Q44. How to implement a method returning an IEnumerable?	240
Q45. What advantages do Collection classes have over Arrays?	245
Q46. Why are there so many Collection classes and which one should I use?	251
Q47. What problem do Concurrent Collections solve?	257
Q48. How are Immutable Collections different from other collections?	263

SECTION VI

LANGUAGE INTEGRATED QUERY (LINQ)

Q49. What is the basic structure of a LINQ query?	268
Q50. How is LINQ query syntax translated to C# method calls?	272
Q51. How to express Inner and Outer Joins using LINQ?	278
Q52. Which Set manipulation operations are available in LINQ?	285
Q53. How can LINQ query results be aggregated?	288
Q54. How can LINQ query results be grouped?	291
Q55. How can LINQ query results be reused without reevaluating?.....	295
Q56. When and how is a LINQ query executed and how does this affect performance?	299
Q57. How are lambda expressions used in LINQ queries?.....	306

Q58. When should my method return IQueryable<T> and when IEnumerable<T>?	310
---	-----

SECTION VII

PARALLEL AND ASYNCHRONOUS PROGRAMMING

Q59. What are the differences between Concurrent, Multithreaded and Asynchronous programming?	316
Q60. Which Timer class should I use and when?	319
Q61. How to create a New Thread and manage its Lifetime?	323
Q62. What abstractions for Multithreaded programming are provided by the .NET framework?	327
Q63. What is the recommended Asynchronous Pattern in .NET? ...	330
Q64. What is the Asynchronous Programming Model?	335
Q65. What is the Event-based Asynchronous Pattern?	339
Q66. In what order is code executed when using async and await?	343
Q67. What is the best way to start using Asynchronous methods in existing code?	348
Q68. What are some of the Common Pitfalls when using async and await?	353
Q69. How are Exceptions handled in Asynchronous code?	358
Q70. What are some Best Practices for Asynchronous file operations?	364
Q71. What is a Critical Section and how to correctly implement it?	370
Q72. How to manage multiple threads in the .NET framework?	375

SECTION VIII

SERIALIZATION & REFLECTION

Q73. What types of Serialization are supported in the .NET framework?
383

Q74. What is Reflection and what makes it possible?394

Q75. What are the potential dangers of using Reflection?400

Q76. When to create Custom Attributes and how to inspect them at
runtime?406

SECTION IX

C# 6 AND 7

Q77. How has C# changed since its first version?412

Q78. Which C# 6 features should I really start using?428

Q79. How did tuple support change with C# 7?433

Q80. What are Local functions and when can they be useful?438

Q81. What is pattern matching and how is it supported in C# 7? ...445

Q82. What is a Discard and when can it be used?.....450

Q83. What improvements for Asynchronous programming have been
introduced in C# 7?454

Q84. How was support for passing values by reference expanded in C# 7?
457

Q85. How to handle errors in recent versions of C#?463

SECTION X

A LOOK INTO THE FUTURE

Q86. What Functional programming features are supported in C#?470

Q87. What are some major new features in C# 8?478

Q88. How are .NET Core and .NET Standard evolving?487

SECTION I

.NET and the Common
Language Runtime (CLR)

1

Q1. Which platforms can I develop applications for using the .NET framework?

When Microsoft released the first version of .NET framework in 2002, the applications developed on it could only run on Windows machines with the .NET Framework installed. This has not changed until today. Applications developed using the **full .NET framework** will still run only on Windows.

However, since then Microsoft has released many new .NET implementations, which targeted other platforms too, while still allowing development using C#, and a very similar base class library.

The first such alternative .NET implementation was **Silverlight**, released in 2007.

Silverlight was a browser plugin for Windows and macOS browsers, similar to the [Adobe Flash Player](#). Developers could design user interfaces in XAML, just like WPF desktop applications. With version 2 released in 2008, Silverlight also included a subset of the .NET framework class libraries. This made it easy for experienced .NET developers to use it for developing cross platform rich internet applications. However, with the decline of browser plugins in favor of HTML and JavaScript based

applications, its popularity decreased. Its last major version was Silverlight 5, released in 2011. Today Silverlight applications are still [supported only in Internet Explorer on Windows](#).

With the release of Windows Phone 7 in 2010, Microsoft released a new .NET implementation based on Silverlight. Again, XAML was used for designing the user interface and this release included a subset of the .NET framework class libraries. The end result was more similar to Silverlight than to the full .NET framework. **Windows Phone Silverlight** (as it is now referred as) was the main framework for developing Windows Phone applications up to and including Windows Phone 8. Applications developed on it still works on the latest version of Windows Phone, i.e. Windows 10 Mobile Fall Creators Update, but they don't have access to the new features of the operating system.

In 2012, Windows 8 introduced Windows Runtime (or WinRT for short) – a new set of APIs for Windows applications, including a XAML based UI framework. When consumed from C#, [a subset of full .NET framework](#) class libraries could be used with it as well. Applications written in WinRT can run on Windows 8 or later, on Windows Phone 8.1, on Windows 10 Mobile, and on the now discontinued Windows RT operating system for ARM-based devices. They run in a sandboxed environment even on desktop computers, restricting direct access to the file system and [communication with other applications](#). They are distributed through Microsoft Store (formerly named Windows Store) and need to go through an approval process before they are published. The current name for this development platform is **Windows Store**.

The next version of WinRT was named **Universal Windows Platform (UWP)** and was released along with Windows 10 in the year 2015. Applications written on it could run on even more platforms: Windows 10, Windows 10 Mobile, Windows 10 IoT Core ([targeting Raspberry Pi 2 or 3 and similar devices](#)), Windows 10 Holographic (running on Microsoft HoloLens and other mixed reality headsets), and Xbox One. UWP is based on WinRT and supports an ever increasing [subset of .NET class libraries](#) and XAML for user interface design. Applications are still run in a sandbox and distributed through Microsoft Store.

The latest .NET implementation by Microsoft is **.NET Core**, first released in 2016.

.NET Core is an open source re-implementation of the full .NET framework, available for Windows, macOS and Linux. It provides a subset of the .NET class libraries, which grew much larger with the release of .NET Core version 2.0 in 2017. It supports development of console applications and ASP.NET Core based web applications for all

supported operating systems. .NET Core Version 3.0 (planned for release in 2019) will add support for Windows Forms and WPF based Windows desktop applications. Entity Framework 6.0 will also be supported in .NET Core 3.0.

Independent of Microsoft, programmer Miguel de Icaza, who is now a Distinguished Engineer at Microsoft, developed an open source implementation of the .NET framework for Linux, named **Mono**. Since the release of version 1 in 2004, it is continuously evolving. Today it supports a large subset of .NET class libraries, including Windows Forms for desktop application development and ASP.NET for web application development. It runs on a large number of operating systems, including Linux, macOS, Sun Solaris, BSD, Windows, and even gaming consoles, such as Sony PlayStation 4 and Xbox One. Earlier versions of Mono also offered support for Nintendo Wii and Sony PlayStation 3. An important complement to the gaming console support in Mono was MonoGame, an open source reimplement of the abandoned Microsoft XNA Framework for game development, originally supported on Windows, Windows Phone and Xbox 360. With Mono, MonoGame games can now run on most computers and operating systems.

Xamarin is a .NET implementation based on Mono, which supports the development for iOS, Android and macOS. It was developed by Miguel de Icaza's company of the same name. Microsoft acquired Xamarin in 2016. Its development model is based on a subset of .NET class libraries, combined with the ability to invoke native APIs of each supported platform. Additionally, Xamarin.Forms, a XAML based user interface framework can be used for development of common user interfaces for iOS, Android and Windows Mobile applications.

Mono is also used in **Unity 3D**, a cross platform game development engine, which currently [supports over 25 different platforms](#), including mobile, desktop, gaming consoles, virtual reality and augmented reality platforms. Although the game development mostly takes advantage of the proprietary Unity 3D game engine, the recommended scripting language is C# and takes advantage of a subset of .NET class libraries.

The latest .NET runtime is **Tizen.NET**. It is developed by Samsung and runs on their Linux based Tizen operating system for smartphones, smart TVs, wearable devices and other types of devices. It is based on .NET Core, but also takes advantage of Xamarin.Forms as the user interface framework.

Runtime	Windows	macOS	Linux	Windows Phone	iOS	Android	Tizen	Gaming consoles	Windows Holographic	Remarks
.NET Framework	Yes									Any application model (web, desktop...)
Silverlight	Yes	Yes								Browser plugin
WP Silverlight				Yes						Sandboxed applications
Windows Store	8+			8.1+						Sandboxed applications
UWP	10+			10+				Xbox One	Yes	Sandboxed applications
.NET Core	Yes	Yes	Yes							Web and console applications only (and Windows desktop applications in v3.0+)
Mono	Yes	Yes	Yes					Yes		.NET framework subset
Xamarin		Yes			Yes	Yes				Access to native APIs
Unity 3D	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Game engine
Tizen.NET							Yes			Applications for smart devices

Table 1: Overview of supported platforms for each .NET implementation

Taking into account all of the listed implementations, .NET framework can be used today to develop at least some types of applications for most of the actively developed platforms and operating systems.

2

Q2. How is .NET Core different from the .NET framework?

The .NET Framework comes to mind when talking about .NET. However, since the release of .NET Core 1.0 in 2016, we need to be more specific and distinguish between the .NET framework (also referred to as the full .NET framework) and the new .NET Core.

To understand the differences between the two, we first need to inspect the two main parts of the .NET framework: Common Language Runtime (CLR) and the Framework Class Library (FCL).

The Common Language Runtime is the virtual machine part of the .NET framework, responsible for running applications. It [provides multiple services](#) which make the following possible:

- The just-in-time (JIT) compiler converts the compiled IL (intermediate language) code from .NET application binaries packaged as assemblies, into machine code for the CPU on which it is currently running.
- The garbage collector (GC) handles memory management. .NET applications need

not allocate and release memory themselves. The garbage collector automatically releases all the unused objects from memory.

- Thread management allows applications to use multiple threads to improve their scalability and performance.
- Exception handling provides the structure required for throwing exceptions as well as the hierarchical handling of exceptions in applications.
- The Common type system (CTS) is the basis for interoperability between the components developed in different programming languages and compiled into the same IL code.

The Framework Class Library is the toolset for effectively building .NET applications. At its core is the Base Class Library (BCL) – a set of low-level classes for common operations, such as math, string manipulation, I/O operations etc. On top of it are other frameworks, specific to building different types of applications: Windows Forms and Windows Presentation Foundation (WPF) for desktop applications, ASP.NET Web Forms and ASP.NET MVC for building web applications, Windows Communication Foundation (WCF) and ASP.NET Web API for building web services, etc.

Since version 1.0, [three updates of Common Language Runtime were released](#) (1.1, 2.0 and 4.0) as well as several .NET framework updates (3.0, 3.5, 4.5...) without its own CLR version, which mostly expanded and updated the Framework Class Library.

.NET Core is a complete reimplement of the .NET framework. Microsoft designed it with several differences compared to the .NET framework:

- Unlike the .NET framework, which was only built for the Windows operating system, .NET Core CLR is cross-platform, with implementations available for Windows, Linux and macOS.
- It is fully open source, licensed under MIT and Apache 2 licenses. It is developed in the public and accepts external contributions. This gives it the potential of supporting even more operating systems and CPU architectures in the future.
- The .NET framework can only be installed as a component of the operating system, allowing only two versions of the .NET framework to be present on a machine (one for CLR 2.0 and one CLR 4.0). Updates to any of the versions affect all installed .NET applications. On the other hand, .NET Core can be deployed as a part of the application itself. This allows different versions of .NET Core to

be installed side-by-side, making it possible for an application to keep running on the same version of the runtime, even if the runtime is updated for other applications.

- The development tools for .NET Core are not tightly bound to Visual Studio. All operations can be performed using command line tools. This makes it easier for other editors and integrated development environments (IDEs) to provide some level of support for .NET Core development, and also gives more control to developers for automating different steps in the development process according to their needs and preferences.
- A lot of attention is given to performance characteristics of the runtime. In the age of microservices and containers, small footprint and quick startup times are much more important than they used to be. Great performance is a prerequisite for .NET Core to be competitive with other frameworks like Node.js or Django.

Although [the assembly formats](#) are compatible between the .NET framework Common Language Runtime (CLR) and the .NET Core CLR, .NET framework applications still can't simply be run under .NET Core. Not only do the entry assemblies need to be recompiled to target a different runtime, but there are also differences in the class libraries.

.NET Core includes only a subset of the .NET framework's Framework Class Library. Although the class library size has more than doubled in .NET Core 2.0, it is still mostly limited to the BCL only.

.NET Core includes only a very limited number of application frameworks:

- ASP.NET Core is a re-implementation of ASP.NET MVC and ASP.NET Web API from the .NET framework. Although the basic concepts are still the same, existing applications can't simply be migrated to the new framework. They need to be rewritten. ASP.NET Core is also available for the .NET framework, so that applications written in it can theoretically run on either runtime.
- The only other currently supported application type are console applications. Old .NET framework console applications can be recompiled for .NET Core without changes as long as they do not have other dependencies, which are not available in .NET Core.
- When .NET Core 3.0 gets released in 2019, support will be added for Windows

Forms or Windows Presentation Foundation (WPF) based Windows desktop applications. Both application models will support hosting of Universal Windows Platform (UWP) controls including modern browser and media player components. .NET Core implementations of all three UI frameworks are open sourced, similar to the rest of .NET Core.

	.NET Framework	.NET Core
Supported OS	Windows	Windows, Linux, macOS
License	closed source	MIT, Apache 2
Install location	system-wide	system-wide or as part of application
Application frameworks	Console, ASP.NET Web Forms, ASP.NET MVC, WinForms, WPF, WCF...	Console, ASP.NET Core (and WinForms, WPF on Windows in v3.0+)

Table 1: Differences between the .NET framework and .NET Core

The current state of .NET Core therefore makes it suitable for writing new web applications and REST services using ASP.NET Core. Unless they have very specific dependencies which are still not supported in .NET Core and can't be moved into standalone services, the benefit of migrating to .NET Core will be better performance and the ability to run the application on operating systems other than Windows.

The same can probably be said for console applications. For any other types of desktop applications though, .NET framework is still the only available option. At least, until the release of .NET Core 3.0.

3

Q3. What is .NET Standard and why does it matter to me?

Since the original release of the .NET framework in 2002, the .NET ecosystem expanded with many alternative .NET runtimes: .NET Core, Universal Windows Platform (UWP), Xamarin, Unity Game Engine and others. Although the runtimes are similar to each other, there are some differences between their base class libraries (BCL), which makes it difficult to share code between projects targeting different runtimes.

Microsoft's first attempt at solving this problem were Portable Class Libraries (PCL), which allowed building assemblies that could be used with multiple runtimes without recompiling. Portable Class Libraries require the target runtimes to be selected when creating the library as based on this choice, the intersection of available APIs is calculated.

With an increasing number of runtimes and different versions, this approach was becoming ever more difficult to manage. Hence, Microsoft introduced .NET Standard as an alternative for easier cross-platform development. .NET Standard takes a reversed approach to determining the available set of APIs. Instead of calculating them based on the targeted runtimes, it explicitly specifies the APIs which must be supported by a

compatible runtime. Several different versions of .NET Standard are defined, each with an increasing number of supported APIs.

Any class library author can now select a .NET Standard version to target their library. A library targeting a specific .NET Standard version will work on any platform implementing this version of the standard. If a platform is updated to support a higher version of .NET Standard or if a new platform is released, all existing libraries targeting the newly supported version of .NET Standard will automatically work with it with no changes required by their respective authors.

Different versions of existing runtimes support different versions of .NET Standard.

Target Platform	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
UWP	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299
Windows Store	8.0	8.0	8.1					
Windows Phone	8.1	8.1	8.1					
Windows Phone Silverlight	8.0							

Table 1: Supported runtimes for different .NET Standard versions (version 10.0.16299 of UWP was shipped as part of Windows 10 Fall Creators Update in October 2017)

What might not be obvious from the table is the big difference between .NET Standard 1.x and .NET Standard 2.0.

Compared to .NET Standard 1.6, .NET Standard 2.0 includes over 20,000 additional APIs. These are APIs that are already a part of the .NET framework, but have been missing from .NET Standard before.

Since .NET Standard is focused on cross-platform compatibility, it still doesn't include all the .NET framework APIs, and it never will. By design, it excludes all application model APIs, such as WPF, Windows Forms, ASP.NET and others. It also excludes all Windows specific APIs that were included in the .NET framework, e.g. APIs for working with the registry and event logs. Most of the .NET framework APIs that do not fall in

these two categories are included in .NET Standard 2.0.

Some of the most notable additions that were missing from .NET Standard 1.6 are:

- System.Data namespace with DataSet, DataTable, DataColumn and other classes for local data manipulation that were commonly used in earlier versions of the .NET framework before the release of Entity Framework.
- XmlDocument and XPath based XML parsers in addition to XDocument, which was the only one available before .NET Standard 2.0.
- System.Net.Mail, System.Net.WebSockets and other previously missing network related APIs.
- Low level threading APIs such as Thread and ThreadPool.

There's also a namespace that's not in .NET Standard, but is worth mentioning as it might come as a surprise to most developers: **System.Drawing**.

The reason this namespace is not included in .NET Standard is because it is just a thin layer over the native GDI+ Windows component and therefore can't be easily ported to other runtimes. This is unfortunate as it is probably the most commonly used .NET framework API that's still missing from .NET Standard. There are currently no plans to include them in a future version of .NET Standard.

Although the System.Drawing APIs can't be used on all .NET Standard compliant runtime, they are made available to .NET Core as part of the [Windows Compatibility Pack](#). Despite the name of this pack, many of its APIs aren't just supported on Windows, but on other platforms too. APIs from the System.Drawing namespace count among them and can also be used on Linux and macOS since .NET Core 2.1.

The increased API set in .NET Standard 2.0 makes it much easier to compile existing source code that was originally written for the .NET framework, and make it available to other runtimes. At least as long as it is not dependent on particular application models. This means that business logic could be shared across runtimes, but applications will still need to be written for each runtime.

However, our own source code, even with business logic, often depends on other third-party libraries, in addition to the base class library APIs that are now exposed via .NET Standard. As expected, .NET Standard libraries can reference other .NET Standard libraries that target the same or lower version. They can also reference compatible

portable class libraries.

Unfortunately, a large majority of existing third-party libraries do not target .NET Standard or PCL, but the .NET framework. Although many of those might be recompiled for .NET Standard with minimal or no code changes at all, this still needs to be done by their authors who might not actively maintain them anymore.

To avoid that requirement altogether, .NET Standard 2.0 includes a *special compatibility shim*, which makes it possible to reference existing .NET framework libraries from .NET Standard libraries, without recompilation. To allow that, any references to the .NET framework classes from the referenced .NET framework library are type forwarded to their counterparts in .NET Standard. Hence, the compiled code will also work on other runtimes, not only on the .NET framework.

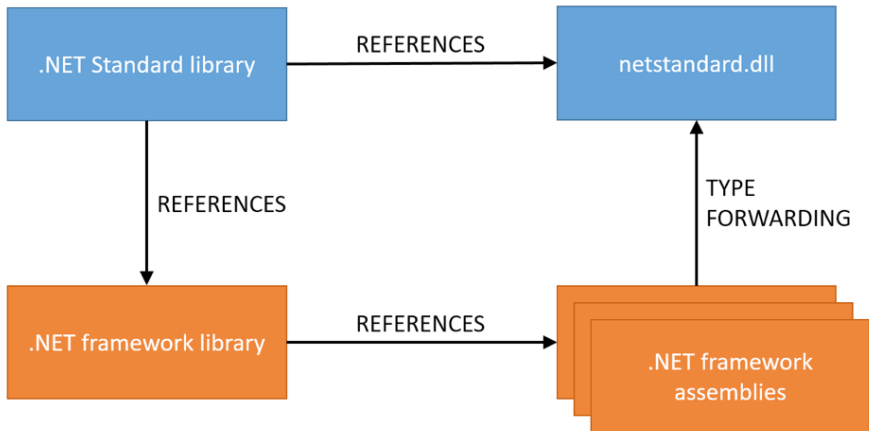


Figure 1: Type forwarding for referenced .NET framework libraries

Of course, this doesn't mean that any .NET framework library will just work with .NET Standard. If it uses any APIs that are missing from .NET standard, the call will fail.

To estimate the severity of this issue, Microsoft did an analysis of all the libraries available on NuGet before releasing .NET Standard 2.0 and published the results in a [video by Immo Landwerth](#). According to it, over two-thirds of libraries were API compatible with .NET Standard 2.0, i.e. they either targeted .NET Standard or PCL, or they targeted the .NET framework but only used APIs that are part of .NET Standard 2.0.

Almost two thirds of those that were incompatible, depended on APIs in specific application models (WPF, ASP.NET, etc.) and were therefore not appropriate for use from .NET Standard. The rest mostly used APIs that were already considered for .NET

Standard 2.0 but were excluded as ‘too hard’ to implement across all runtimes. It is therefore not expected that compatibility will significantly increase with future .NET Standard versions.

Compatible packages (68 %)		Incompatible packages (32 %)	
.NET framework (56 %)	PCL, .NET Standard (12 %)	Unsupported application models (20 %)	Missing APIs (12 %)

Table 2: Assembly compatibility in NuGet

A small number of API compatible libraries from the above analysis (less than 7%) used P/Invoke to call into native libraries. Although these assemblies can be used from .NET Standard, they will only work on Windows (when referenced from the .NET framework or .NET Core applications) but will fail on other operating systems because they depend on native binaries.

4

Q4. How can I create a .NET Standard library and use it from different runtimes?

Creating a .NET Standard class library is the best way for sharing common code between applications which target different runtimes, e.g. a desktop application in the .NET framework, an ASP.NET Core web application in .NET Core, and an Android or iOS application in Xamarin. The compiled class library will be binary compatible with all current and future runtimes which support the targeted version of .NET Standard.

To create a .NET Standard library in Visual Studio 2017, start by creating a new project based on the **Class Library (.NET Standard)** project template. To see which version of .NET Standard is targeted, and to change it, you can check the **Application** tab of project properties.

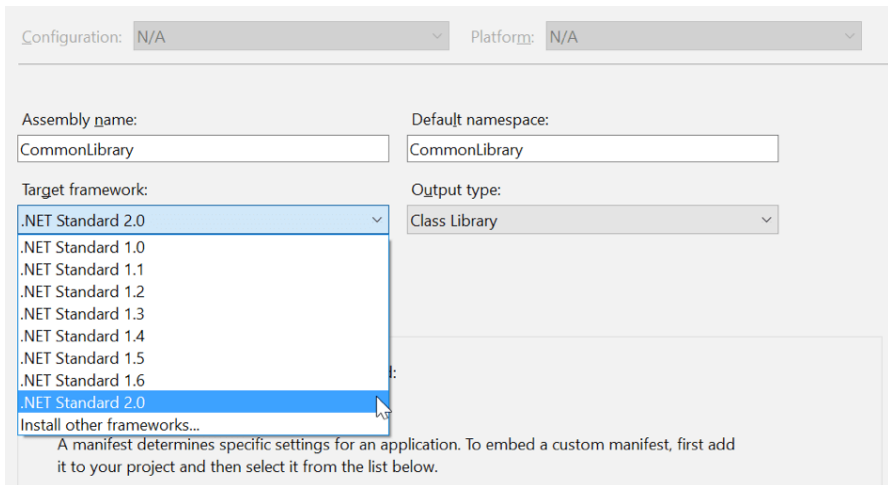


Figure 1: .NET Standard version selection in project properties

The minimum requirements for .NET Standard 2.0 are Visual Studio 2017 15.3 and .NET Core SDK 2.0. If you don't have these two installed on our computer, .NET Standard 2.0 won't be available in the dropdown that you see in Figure 1.

Although your first impulse might be to always target the latest version of .NET Standard, you should base your decision on your requirements:

- The higher version of .NET Standard you target, the more APIs will be available to you. You can use the [.NET API Browser](#) to check which APIs are supported in which version of .NET Standard.
- The lower the version of .NET Standard you target, the more the runtimes you will be able to use your class library in. You can always find up-to-date information on which runtime version supports which version of .NET Standard in the [official .NET Standard documentation](#).

To support as many runtimes as possible without imposing additional restrictions on what APIs you may use, choose the lowest version of .NET Standard which includes all the APIs you require.

To try out how the common .NET Standard library can be consumed from different runtimes, let's create a class in our library with a simple method in it:

```
namespace CommonLibrary
{
    public class Messages
    {
        public string Hello(string name = "World")
        {
            return $"Hello {name} from .NET Standard!";
        }
    }
}
```

It is important to understand that .NET Standard is not a runtime, therefore we cannot create a .NET Standard application of any kind. Only class libraries can target .NET Standard. When creating an application, you need to decide what application type you want to create, and select a suitable runtime.

Let us first try to consume the library from a classic desktop .NET framework application. Add a new project to the solution in Visual Studio based on the **Console App (.NET Framework)** project template. Here we can add a project reference to our common .NET Standard library as if it were a .NET framework class library. However, make sure that the targeted version of .NET framework supports the version of .NET Standard used in our common library. For example, to reference a .NET Standard 2.0 class library, the application must target .NET framework 4.6.1 or higher. If that's not the case, you will get a build error:

Project '..\CommonLibrary\CommonLibrary.csproj' targets 'netstandard2.0'. It cannot be referenced by a project that targets '.NETFramework,Version=v4.6'.

We can select the target framework when creating the project. To change it afterwards, there is a **Target framework** dropdown in the **Application** tab of project properties.

Let's now print out the message from the common library to the console:

```
using CommonLibrary;
using System;

namespace NetFrameworkApp
{
    class Program
    {
```



```

static void Main(string[] args)
{
    var messages = new Messages();
    Console.WriteLine(messages.Hello(".NET Framework"));
    Console.ReadLine();
}
}
}

```

As expected, the following message will be written to the console when you run the application:

Hello .NET Framework from .NET Standard!

To consume the common .NET Standard library from a .NET Core application, you need to follow similar steps. Add a new project to the solution based on the **Console App (.NET Core)** project template. Again, add the .NET Standard library as a project reference to the project. In this case we will get two errors if the targeted .NET Core version doesn't support the library's version of .NET Standard.

Project CommonLibrary is not compatible with netcoreapp1.1 (.NETCoreApp,Version=v1.1). Project CommonLibrary supports: netstandard2.0 (.NETStandard,Version=v2.0)

Project '..\CommonLibrary\CommonLibrary.csproj' targets 'netstandard2.0'. It cannot be referenced by a project that targets '.NETCoreApp,Version=v1.1'.

.NET Standard 2.0 is only supported from .NET Core 2.0. When creating a new .NET Core project in Visual Studio 2017, the highest available version will be used by default. It can be changed by using the **Target framework** dropdown in the **Application** tab of project properties.

The code for our console application will remain almost identical:

```

using CommonLibrary;
using System;
namespace NetCoreApplication
{
    class Program
    {

```

```
static void Main(string[] args)
{
    var messages = new Messages();
    Console.WriteLine(messages.Hello(".NET Core"));
    Console.ReadLine();
}
}
```

It shouldn't come as a surprise that the application will write the following message to the console:

Hello .NET Core from .NET Standard!

There is an important difference between the two applications though. The first one targets .NET framework and only works on Windows. The second one targets .NET Core and will also run on Linux and macOS.

You can use the same approach to consume a common .NET Standard library from any other runtime that also supports its version of .NET Standard. To take full advantage of sharing common code between the runtimes, we won't be putting only messages there. .NET Standard 2.0 includes a large part of the .NET framework APIs, allowing us to share most business logic code.

For example, you can put data transfer objects (DTOs), data validation code, application specific calculations etc. into the common library and use it in the web application on the server, and in desktop and mobile applications. The obvious benefit is having only a single piece of code that needs to be maintained.

With .NET Standard 2.0, you can also reference .NET framework class libraries from applications written in any runtime which supports .NET Standard 2.0. These libraries will work as if they were .NET Standard libraries as long as they don't use any unsupported APIs. This is most useful when we need to use third-party libraries which weren't recompiled for .NET Standard, but you can try this out with your own .NET framework library as well.

Let's add another project to the solution, this time based on **Class Library (.NET Framework)** project template. We'll use the same code as in the .NET Standard library:

```
namespace NetFrameworkLibrary
{
    public class Messages
    {
        public string Hello(string name = "World")
        {
            return $"Hello {name} from .NET Framework!";
        }
    }
}
```

Now, you can reference this library from the .NET Core console application instead of the .NET Standard library and modify the code accordingly:

```
using NetFrameworkLibrary;
using System;
namespace NetCoreApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            var messages = new Messages();
            Console.WriteLine(messages.Hello(".NET Core"));
            Console.ReadLine();
        }
    }
}
```

The application will still build and run. The modified message will be written to the console:

Hello .NET Core from .NET Framework!

This only works in .NET Core 2.0, though. If we try the same from an earlier version of .NET Core, we will get a somewhat cryptic build error:

The type 'Object' is defined in an assembly that is not referenced. You must add a reference to assembly 'mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.

This happens because we are trying to run .NET framework code in .NET Core runtime. .NET Core 2.0 implements the type forwarding necessary to map .NET framework classes to corresponding .NET Core classes. There's no such type forwarding present in .NET Core 1.1, hence the build fails because of a missing type that's referenced in the class library.

5

Q5. What is the Roslyn compiler and how can I use it?

Ever since its first public preview in 2011, *Roslyn* has been known as the code name for the next generation C# compiler. Although, the project had started internally at Microsoft a couple of years earlier. Even before its first final release in Visual Studio 2015, it was evident that Roslyn was much more than just a new compiler. At that time, it also got a new official name: **.NET Compiler Platform**. Nevertheless, the word Roslyn is still a part of developer vocabularies and will probably remain in use for quite some time.

Microsoft decided to rewrite the C# and Visual Basic compilers from scratch, because their code base became a challenge to maintain, and adding new features became unfeasible. However, they did not want to just improve the code, but also to make the compiler useful in other scenarios as well: diagnostics, static analysis, source code transformation, etc. In order to achieve that, they created a compiler that not only converts source code into binaries, but also acts as a service, providing a public API for understanding the code.

Roslyn APIs reflect the pipeline architecture of a traditional compiler, giving access to

each step of the compiler's source code analysis and processing:

- **Syntax tree API** exposes the lexical and syntactic structure of the code, including formatting and comments. The latter two are irrelevant for later phases of the compiler pipeline, but important for tools that want to manipulate the code and keep the formatting and code comments intact.
- **Symbol API** exposes the symbol table containing names declared in the source code, as well as those originating from referenced assemblies without corresponding source code.
- **Binding and Flow Analysis APIs** exposes the complete semantic model of the code that becomes available after the binding phase of the compiler pipeline. These APIs contain all the information about the code that is required for generating the binaries, including any errors and warnings that were detected in the code.
- **Emit API** provides access to services for emitting the IL byte code.

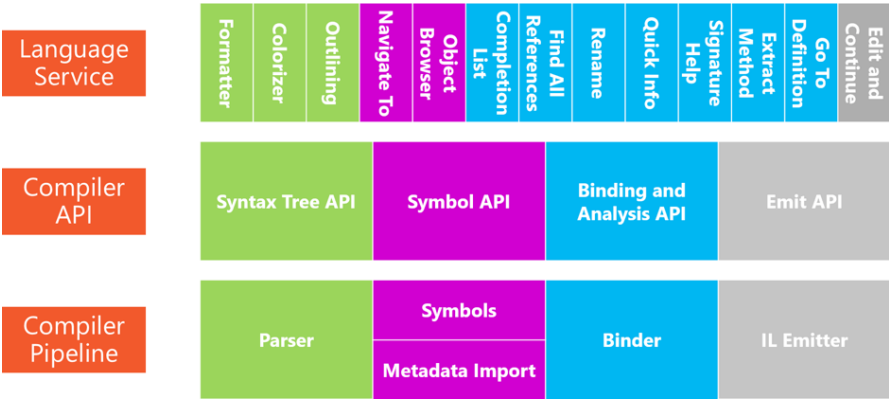


Figure 1: Compiler as a service

The scope of Roslyn is not limited to the compiler API. Most of its integration into Visual Studio is part of the public API as well, except for the thin layer interacting with its proprietary code:

- **Diagnostic API** enables development of custom third-party code diagnostics that are pluggable into the Visual Studio editor and MSBuild build system, making them indistinguishable from the compiler's built-in diagnostics for detecting errors and warnings in code.
- **Scripting API** provides the interactive C# environment in Visual Studio – the so-

called REPL (read-eval-print loop).

- **Workspaces API** provides the model of a complete solution, consisting of multiple projects with documents and assembly references, making it possible to drill down all the way to the semantic and syntactic models of the code.

As a .NET developer, you need not know about all these APIs, nor do you have to understand the inner workings of a compiler. If you're programming in C# 6 or a more recent version (i.e. you are using Visual Studio 2015 or later), you are already using the new Roslyn compiler and thanks to its service architecture, you're not only using it to compile your source code, but also to improve your code editing experience.

Modern integrated development environments, such as Visual Studio, are much more than just a text editor. They offer many additional functionalities: syntax highlighting, code completion, debugging, refactoring, static analysis, etc. These services require some level of knowledge about your source code. Before Roslyn, compilers did not share the intermediate results of their processing, therefore Visual Studio editor had to do a lot of C# language processing independently of the C# compiler.

Now, Visual Studio is taking full advantage of the .NET Compiler Platform. Many of its code editor functionalities are implemented using the Roslyn public API: from automatic code formatting and coloring; to IntelliSense, code navigation and refactoring. Visual Studio is not the only code editor or integrated development environment that's using Roslyn internally: [Visual Studio for Mac](#) and [Visual Studio Code](#) are doing it, too.

Since the full Roslyn public API is available as open source, non-Microsoft text editors can also use it. Except for [MonoDevelop](#), most of them are not written in .NET and therefore can't host Roslyn in-process. Instead, they use [OmniSharp](#) – a cross-platform wrapper around Roslyn exposing its API over HTTP. That's how [Atom](#), [Brackets](#), [Sublime Text](#) and other text editors provide C# language services inside the code editor window.

With the help of Roslyn, code diagnostics and refactoring became accessible to individual developers. Earlier, this used to be possible using commercial extensions like [ReSharper](#) and [CodeRush](#). Now, Visual Studio provides a built-in unified user interface for indicating code errors and warnings with wavy underlines (also known as squiggles), and to provide actions for refactoring the affected code and fixing it in

a dropdown, with a lightbulb icon. All of these UI elements are strongly inspired by those commercial extensions.

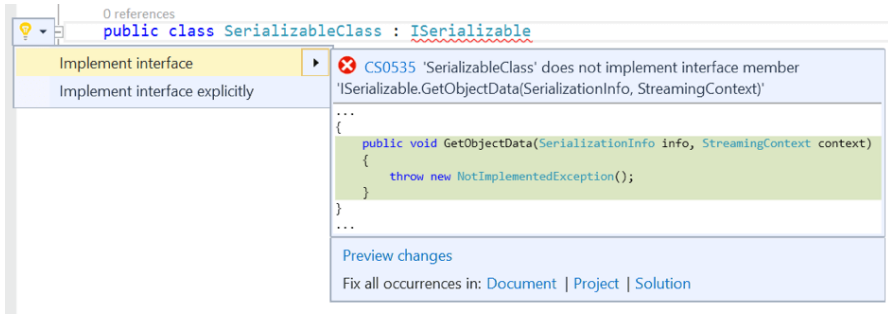


Figure 2: Available code fixes and a preview window with code changes

The user interface is not limited to built-in static analyzers and code fixes, but is also used for displaying results of third party diagnostic analyzers, such as [Refactoring Essentials](#) and [Code Cracker](#), which use the Roslyn API to implement their own static analyzers and refactorings. They can be installed in two ways:

- As Visual Studio extensions, which makes these analyzers available in all projects, but limited to those Visual Studio installations with the extension installed.
- As NuGet packages in the project, which restricts these analyzers to that specific project, but makes them available to anyone who opens the project in their installation of Visual Studio, whether they have the extension installed or not. These analyzers can also be executed at build time, even on the build server.

Since Roslyn APIs do all the hard work of parsing the source code into the abstract syntax trees and generating the changed code after refactoring, writing your own diagnostic becomes much easier if you have at least a basic understanding of compilers. That's why there are so many open source diagnostic analyzers of varying scope and quality available.

You might want to create your own diagnostic analyzer that provides a functionality specific to your code base, e.g. to validate best practices of using your internal APIs or to enforce design or style guidelines. To get you going, there are [Visual Studio project templates](#) available for download with sample code included.

6

Q6. How did the .NET Framework evolve since version 1.0?

It's been over 15 years since the original release of .NET framework 1.0 and looking at it today, it seems much less impressive than it did. Most of the features we are used to, were not available back then and we would have had a hard time if we wanted to use it for development of modern applications today.

This is understandably the case with any framework. To keep up with new technology advancements, development frameworks have to evolve too. The .NET Framework has evolved and grown to keep up with these needs.

Let us take an overview of this evolution.

CLR version	.NET framework version	Most important changes
1.0	1.0	Initial release: Console Applications, Windows Forms, Windows Services, Web Forms, Web Services
1.1	1.1	IPv6, ASP.NET mobile controls, ADO.NET ODBC
2.0	2.0	Generics, edit and continue, 64-bit runtime
	3.0	WPF, WCF, WF
	3.5	LINQ, ASP.NET AJAX, WF services, cryptography
	3.5 SP1	.NET Client Profile, EF
4	4	DLR, PCL, TAP, ASP.NET MVC
	4.5	Async I/O, ASP.NET Web API, ASP.NET Web Pages
	4.5+	64-bit edit and continue, async debugging, HTTP/2, 64-bit JIT

Table 1: .NET framework version history

.NET framework 1.0 included a limited set of application frameworks, and supported the following types of applications:

- Console and Windows Forms applications for the desktop
- Windows Services
- ASP.NET Web Forms based web applications, and
- Web services using the SOAP protocol.

.NET framework 1.1 didn't bring a lot of changes. Although it featured a new version of the CLR, it was to enable side-by-side installation of both versions (v1.0 and v1.1) on the same computer. The most notable new features were support for IPv6, mobile controls for ASP.NET Web Forms and an out-of-the-box ADO.NET data provider for ODBC. Unless you had a specific need for any of these features, there was no reason convincing enough to migrate to the new version.

.NET framework 2.0 was a different story altogether. It fixed the weaknesses identified in the previous two versions and introduced many new features to boost developer productivity. Let's name only the ones with the biggest impact:

- The introduction of **generics** allowed creation of templated data structures that could be used with any data type while still preserving full type safety. This was most widely used in collection classes, such as lists, stacks, dictionaries, etc. Before

generics, all data types stored in the provided collection classes were treated as instances of `System.Object`. After retrieving them from the collection, the code had to cast them back to the correct type. Alternatively, custom collection classes had to be derived for each data type. Generics weren't limited to collections though. Nullable value types were also relying on them.

- Improved debugging allowed **edit and continue** functionality: when a running application was paused in the debugger, code could be modified on the fly in Visual Studio and the execution continued using the modified code. It was a very important feature for existing Visual Basic 6 programmers who were used to an equivalent functionality in their development environment.
- **64-bit runtime** was added. At the time, only Windows Server had a 64-bit edition, and the feature was mostly targeted at server applications with high memory requirements. Today, most of Windows installations are 64-bit and many .NET applications run in 64-bit mode without us even realizing it.

All of these, along with countless other smaller improvements to the runtime and the class library made .NET framework 2.0 much more attractive to Windows developers who were using other development tools. It was enough to make .NET framework, the de facto standard for Windows development.

.NET framework 3.0 was the first version of the .NET framework which couldn't be installed side-by-side with the previous version because it was still using the same runtime (CLR). It only added new application frameworks:

- **Windows Presentation Foundation (WPF)** was the alternative to Windows Forms for desktop application development. Improved binding enabled better architectural patterns, such as MVVC. With extensive theming support, application developers could finally move beyond the classic *battleship-gray* appearance of their applications.
- **Windows Communication Foundation (WCF)** was a new flexible programming model for development of SOAP compliant web services. Its configuration options made it possible to fully decouple the web service implementation from the transport used. It still features one of the most complete implementations of the WS-* standards on any platform, but is losing its importance with the decline of SOAP web services in favor of [REST services](#).
- **Windows Workflow Foundation (WF)** was a new programming model for long

running business processes. The business logic could be defined using a graphical designer by composing existing blocks of code into a diagram describing their order of execution based on runtime conditions.

While WF wasn't widely used, we can hardly imagine the .NET framework without WCF and WPF (and other XAML based UI framework derived from it, such as Silverlight and UWP).

.NET framework 3.5 continued the trend of keeping the same runtime and expanding the class library. The most notable addition was **Language Integrated Query (LINQ)**. It was a more declarative (or functional) approach to manipulating collections. It was inspired by SQL (Structured Query Language) widely used in relational databases. Thanks to the updates to C# and Visual Basic released during the same time, there was even a query syntax available as an alternative to the more conventional fluent API. By automatically converting the query into a tree-like data structure, it opened the doors to many LINQ providers, which could execute these queries differently based on the underlying data structure. Three providers were included out-of-the box:

- LINQ to Objects performs the operations on in-memory collections.
- LINQ to XML is used for querying XML DOM (document object model).
- LINQ to SQL is a simple object-relational mapping (ORM) framework for querying relational databases.

Web related technologies also evolved with this version of the .NET framework:

- **ASP.NET AJAX library** introduced the possibility of partially updating Web Forms based pages without full callbacks to the server by retrieving the updates with JavaScript code in the page. It was the first small step towards what we know today as single page applications (SPA).
- **WF services** exposed WF applications as web services by integrating WF with WCF.
- Managed support for **cryptography** was expanded with additional algorithms: AES, SHA and elliptic curve algorithms.

Looking at the name of **.NET framework 3.5 SP1**, one would not expect it to include features, but at least two of them are worth mentioning:

- **Entity Framework (EF)** was a fully featured ORM framework that quickly replaced

LINQ to SQL. It also heavily relied on LINQ for querying.

- **.NET Client Profile** was an alternative redistribution package for the .NET framework which only included the parts required for client-side applications. It was introduced as the means to combat the growing size of the full .NET framework, and to speed up the installation process on computers which did not need to run server-side applications.

.NET framework 4 was the next framework that included a new version of the CLR and could therefore be installed side-by-side with the previous version. The most notable new additions were:

- **Dynamic Language Runtime (DLR)** with improved support for late binding. This simplified certain COM interop scenarios and made it easier to port dynamic languages to the CLR. IronRuby and IronPython (Ruby and Python ports for the .NET framework) took advantage of that.
- **Portable Class Libraries (PCL)** were introduced for creating binary compatible assemblies that could run on different runtimes (not only the .NET framework, but also Windows Phone and Silverlight).
- **Task-based Asynchronous Pattern (TAP)** was a new abstraction for writing multithreaded and asynchronous applications which hid many of the error-prone details from the programmers.
- **ASP.NET MVC** was an alternative programming model to Web Forms for creating web applications. As the name implies, it implemented the model view controller (MVC) pattern which was gaining popularity on other platforms. It was first released out-of-band, but .NET framework 4 was the first version to include it out-of-the-box.

.NET framework 4.5 was the last major new release of .NET framework with several new important features:

- Support for **asynchronous I/O operations** was added as the new `async` and `await` syntax in C# made it much easier to consume them.
- **ASP.NET Web API** and **ASP.NET Web Pages** were included in the framework for the first time after they were originally released out-of-band. They allowed development of REST services and provided Razor, an alternative syntax for ASP.NET MVC views, respectively.

- PCL was expanded to support **Windows Store** applications for Windows 8.

Also, the .NET Client Profile was discontinued as optimizations to the size and deployment process of .NET framework made it obsolete.

Since .NET framework 4.5, the release cadence increased. The version numbers (4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2) reflected that. Each new version brought bug fixes and a few new features. The more important ones since .NET framework 4.5 were:

- Edit and continue support for 64-bit applications.
- Improved debugging for asynchronous code.
- Unification of different ASP.NET APIs (ASP.NET MVC, ASP.NET Web API and ASP.NET Web Pages).
- Support for HTTP/2.
- Improved 64-bit just-in-time (JIT) compiler for faster startup time.

The focus of new development has now shifted to .NET Core, therefore new major improvements to the .NET framework are not very likely. We can probably still expect minor releases with bug fixes and smaller improvements though.

7

Q7. How does the version of .NET and C# affect compatibility?

Backward compatibility has been an important part of the .NET framework since its original release. With rare exceptions, any application built for an older version of the .NET framework *should* run in any newer version, with no issues.

We can group .NET framework versions into two categories:

- In the first category are .NET framework versions which include a new version of the Common Language Runtime (CLR). Some breaking changes were introduced in these versions which could affect existing applications. To keep all applications working despite that, these .NET framework versions are installed side-by-side, i.e. while installing a newer version of the framework, the previous version is kept intact on your system. This allows applications that were built for it to still use it. There were four versions of CLR released: 1.0, 1.1, 2.0, and 4.
- In the second category are .NET framework versions, without a new version of the CLR. These only expand the class library, i.e. add the new APIs for applications to use. They install in-place and therefore affect existing applications targeting

the corresponding version of the CLR, forcing them to run on the updated .NET framework version.

Since Windows 8, .NET frameworks 1.0 and 1.1 are not supported any more. The operating system comes pre-installed with the version of .NET framework available at the time of release. Newer versions are installed as part of Windows Update. The pre-installed version is based on CLR 4. .NET framework 3.5 (the latest version running on CLR 2.0) can be optionally installed as a Windows feature.

Applications built for .NET framework 4 or newer will run on the pre-installed CLR 4 based .NET framework version. They might still fail to run properly if they use an API from a newer version of the .NET framework that is not yet installed on the target computer.

Applications built for versions of .NET framework based on CLR 2.0, will by default try to run on .NET framework 3.5. If it is not installed on the target computer, the application will fail to run and prompt the user to install .NET framework 3.5. This behavior can be changed by adding a `supportedRuntime` entry to the application configuration file (named `MyApplication.exe.config`, where `MyApplication.exe` is the affected executable filename):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v2.0.50727"/>
    <supportedRuntime version="v4.0"/>
  </startup>
</configuration>
```

By specifying multiple runtimes, they will attempt to be used in the order given. If .NET framework 3.5 is installed on the machine (v2.0.50727 indicates CLR 2.0), the application will use it. Otherwise it will run on the latest .NET framework version installed. Alternatively, to force the application to run using the latest version of .NET framework even if .NET framework 3.5 is also installed side-by-side, list only runtime v4.0 in the configuration file. If the application has none issues with the latest version of .NET framework, this will allow it to take advantage of optimizations in CLR 4 without recompiling it.

To help with the development of applications for the .NET framework version of choice,

Visual Studio features full multi-targeting support.

When creating a new project, there is a dropdown available in the New Project dialog to select the .NET framework version. This will not only affect how the new project will be configured, but will also hide any project templates that are not supported in the selected .NET framework version (e.g. WPF App cannot be created when .NET framework 2.0 is selected because it was only introduced in .NET framework 3.0).

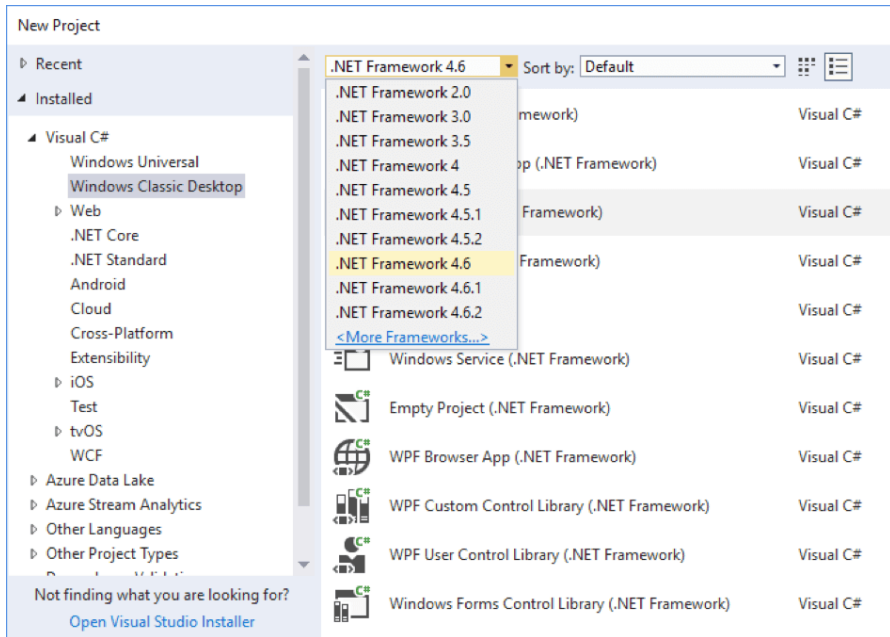


Figure 1: .NET framework dropdown in the New Project dialog

Visual Studio will make sure that only APIs included in the selected .NET framework version will be available to you. If you attempt to use APIs that are not supported, they will be marked accordingly in the code editor.

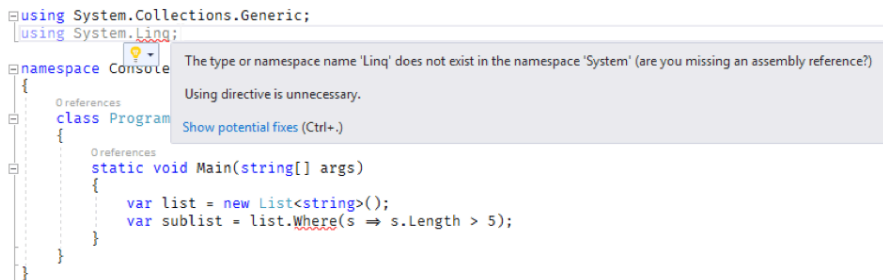


Figure 2: Unsupported APIs marked as errors in the code editor

The target framework for an existing project can be changed at a later time in the *Application* tab of the project properties window. This will have the same effect as selecting the target framework when the project is created. If the existing code used APIs which are not available after changing the target framework, the project will not build until the code is fixed or the target framework is changed back to what it originally was. The compiler will also warn about any referenced system assemblies that are not available so that they can be removed from the project references.

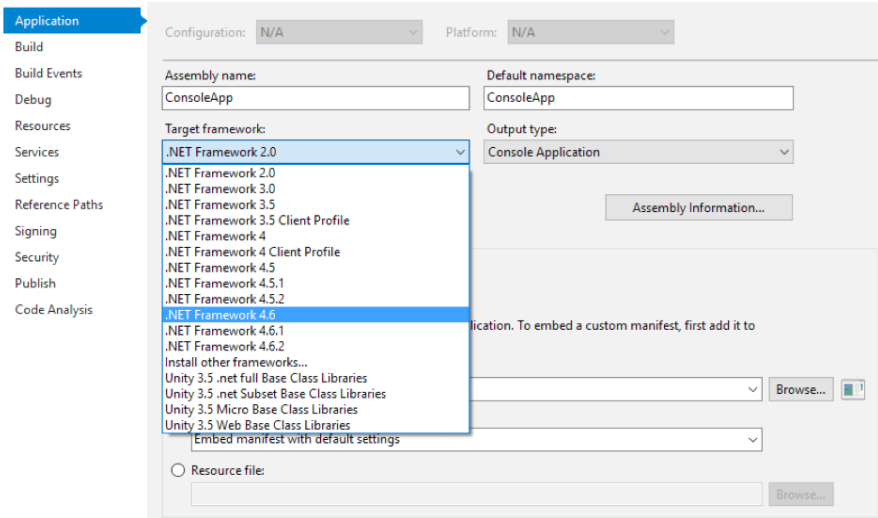


Figure 3: Changing the target framework for an existing project

No matter which .NET framework version we target in the project, the C# language version in use will not change. That's fine because the vast majority of language features that were introduced in later versions of the language don't depend on the CLR or specific APIs. They are only syntactic sugar and the bytecode generated by the compiler will still work in .NET framework 2.0.

There are a couple of exceptions though.

For example, the *async* and *await* keywords are syntactic sugar for the Task-based asynchronous pattern, which was only introduced in .NET 4, therefore they cannot be used in earlier versions of the .NET framework. They also depend on some additional classes, which are only available in .NET framework 4.5. However, these can be added to the project by installing the *Microsoft.Bcl.Async* NuGet package. This way, you can start using *async* and *await* keywords in a project targeting .NET framework 4.

Similarly, some compiler features depend on specific attributes in their

implementation:

- The `System.Runtime.CompilerServices.ExtensionAttribute` is required to compile code taking advantage of extension methods. It was only introduced in .NET framework 3.5.
- Caller information attributes were added only in .NET framework 4.5 and without them we have no way of telling the compiler our intention. However, the code generated by the compiler will still work in older versions of the .NET framework if we add the required attributes with another class library and use those instead.

```
public void LogMessage(string message,
[CallerMemberName] string memberName = "",
[CallerFilePath] string sourceFilePath = "",
[CallerLineNumber] int sourceLineNumber = 0)
{
    // logging implementation
}
```

The compiler requires attributes with the correct fully qualified name to make these features work. To add the caller information attributes to your project, you can install [the official Microsoft.Bcl.NuGet package](#). However, there is no official NuGet package to make extension methods available for older versions of the .NET framework. You can either declare the attribute in your own code or install one of the unofficial NuGet packages.

These limitations don't prevent you from using the latest version of the compiler when targeting an older version of the .NET framework. The code editor in Visual Studio will immediately warn you if you try to use an unsupported feature so you can avoid it while still taking advantage of all the other supported features.

With that being said, there is still a legitimate use case to use an older version of the compiler in your project.

Each compiler version is only available since the Visual Studio version it was originally released with. If some developers on the project are still using an older version of Visual Studio, you will need to avoid using newer language features or they won't be

able to compile the code. To safely achieve that, you can specify the language version in the Advanced Build Settings dialog, accessible via the Advanced... button in the Build tab of the project properties window.

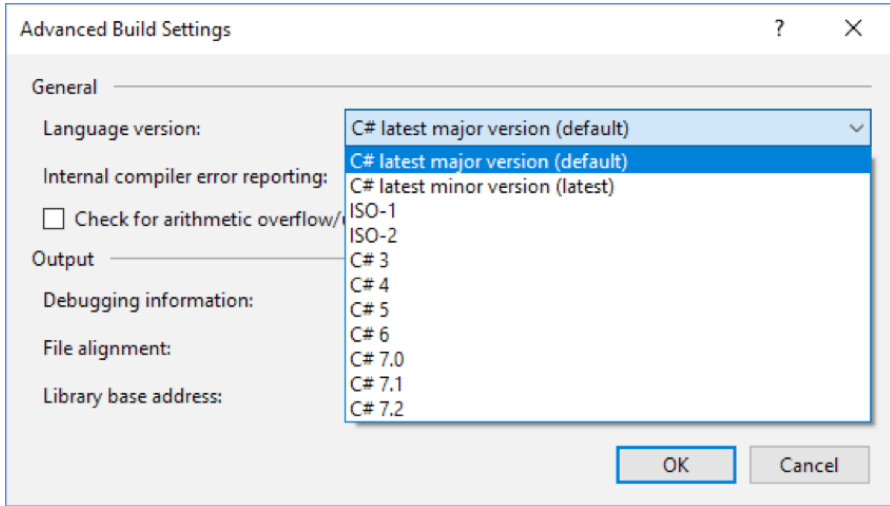


Figure 4: Language version selection in the Advanced Build Settings dialog

By doing so, you will get warnings in the code editor when you try to use a language feature which isn't available in the selected language version. This will allow developers with older versions of Visual Studio to still compile the code without errors.

8

Q8. What is the best way to consume third party libraries?

When creating a large application, not all code will be inside a single project. If you want to reuse the code in multiple applications (e.g. common business logic in desktop and web applications) you can extract it into a separate class library project. As long as the project resides in the same solution, you can reference it as a project and Visual Studio will correctly handle the dependency between the two projects on its own, i.e. when code in either project changes, it will rebuild the projects as necessary.

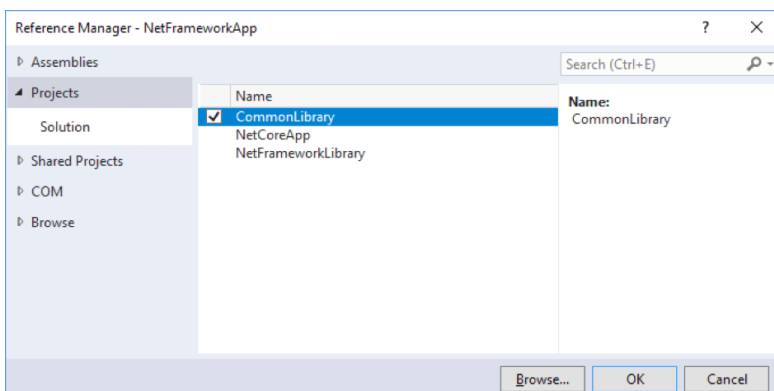


Figure 1: Adding a project reference in Visual Studio

However, usually we won't write all the code for our application ourselves. The project will not only reference our own class libraries and the framework class libraries but also other third-party libraries which will be distributed as compiled assemblies, even if they are open source and their source code is freely available on GitHub. We could directly download a ZIP archive with the assemblies from the library home page, extract the binaries to our local hard drive and reference them directly from our Visual Studio project.

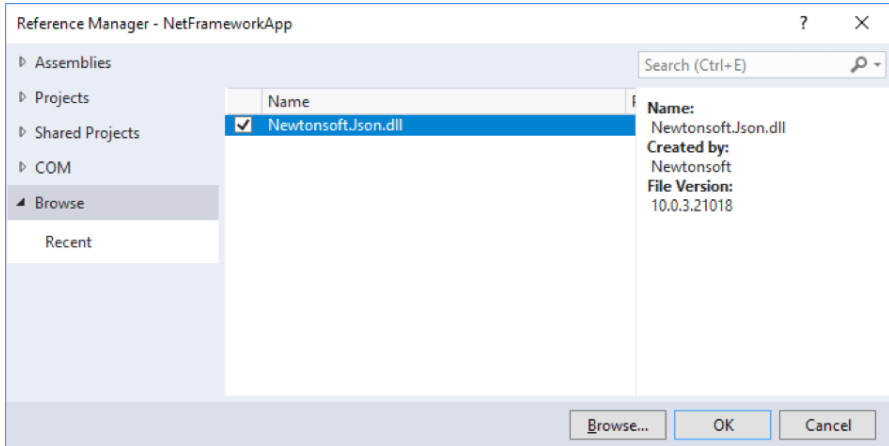


Figure 2: Adding an assembly reference in Visual Studio

However, as soon as we are not the only developer on the project or we start using source control, it becomes important where the assemblies are referenced from. Unless we want every developer to have the assemblies in the same location, we will need to put them somewhere inside the solution folder and add them to source control. We will also have to manually check for new versions of the library and update the assemblies with newer ones. Although source control will show the two assemblies as different binaries, there will be no easy way to see the old and the new version number. If referenced assemblies will have dependencies of their own, it will complicate matters even further.

To help with these challenges, most development platforms today provide a dedicated tool for handling third-party libraries, i.e. a package manager. The package manager for the .NET framework is called NuGet. It was originally released as a Visual Studio extension in 2010. Its functionalities are also built into Visual Studio since Visual Studio 2015. The key part of **NuGet** is a central repository at www.nuget.org/packages where third-party libraries are published and ready to discover. We will usually interact with it through the NuGet Package Manager window in Visual Studio.

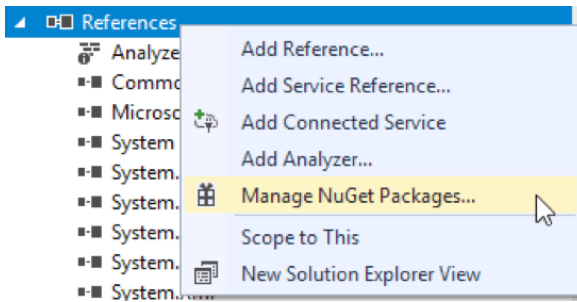


Figure 3: Opening NuGet Package Manager window for a project

The window will list all the installed packages in the project and will allow us to search for a package and install it. As a result, the package will be downloaded locally and a reference will be added to one or more assemblies inside it from our project. When different assemblies are required for different .NET runtimes and their versions, all of them are included in the same package and the right one will be automatically referenced from the project.

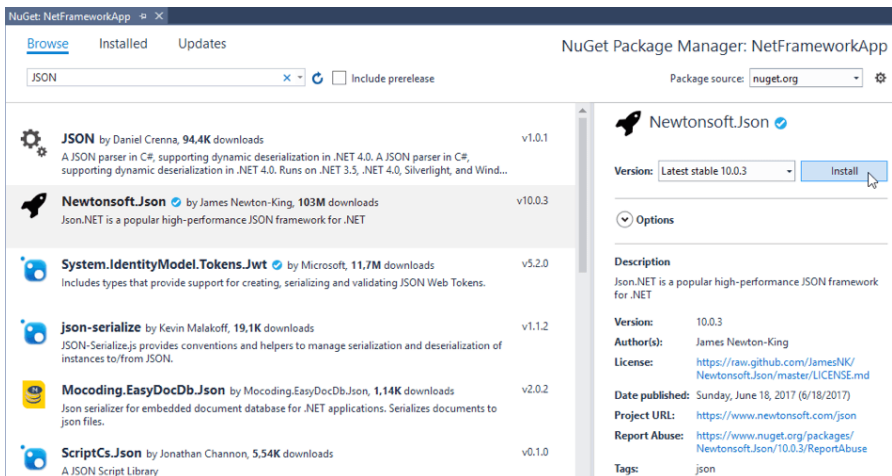


Figure 4: Searching for and installing a NuGet package

For each package installed, we can quickly see which version is installed, and which version is the latest one available (optionally including prerelease versions). We can select a different version (newer or older) and update the package. This will automatically download the correct version of the package and change the reference in the project to point to the correct files.

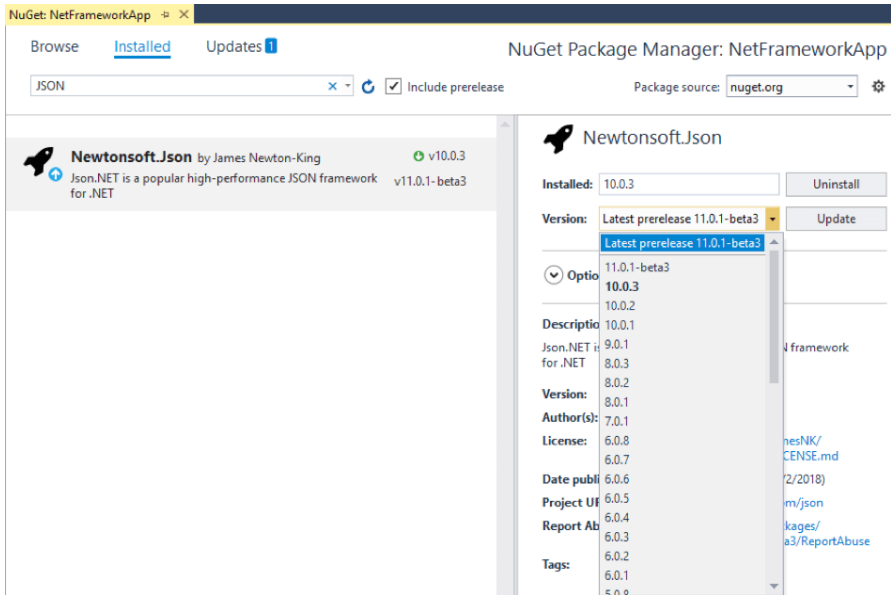


Figure 5: Changing the version of an installed package

All of this works even when packages depend on other packages. The NuGet Package Manager will automatically install all the dependent packages (and recursively all their dependencies as well) when we install such a package. If we install a different version later, NuGet Package Manager will change the versions of dependent packages according to how the package author has defined them in the package manifest. Automatically the dependent packages installed will be included in the list of installed packages.

If we change a version of one of those dependent packages manually, the versions of their dependers will be changed to preserve compatibility. If that is not possible because there is no compatible depender package available, the version of dependent package will not be changed, and the developer will be informed about the problem.

While most of third-party .NET framework libraries are available in the central NuGet repository, some commercial ones aren't. For such cases, you can configure additional package sources for the NuGet Package Manager in the Visual Studio Options dialog. These alternative package repositories can be provided by library vendors, or hosted privately in larger software development companies with common class libraries used across many products and distributed as NuGet packages.

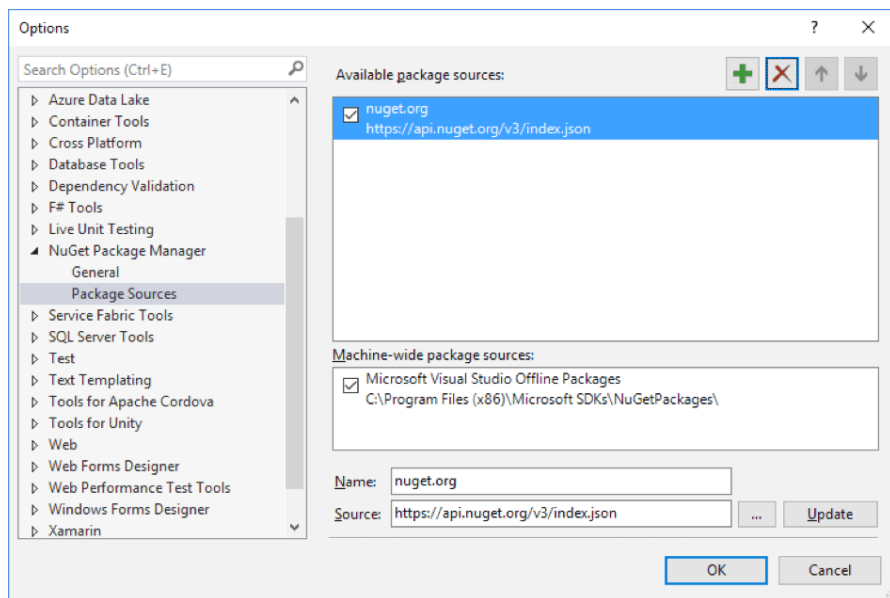


Figure 6: Package Sources configuration in the Visual Studio Options dialog

There's no need to add installed NuGet packages to source control, as they would only make the repository larger. The NuGet Package Manager will write the information about the installed packages to the `packages.config` file inside the project folder:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Newtonsoft.Json" version="10.0.3"
targetFramework="net461" />
</packages>
```

At the start of a build, the NuGet Package Manager will make sure that all listed packages are available locally. If any of them are missing, it will automatically download them. This process is called "package restore". To ensure that all developers are using the same version of all packages, this file is the only one that needs to be added to source control along with your project.

In .NET Core projects, this file is not necessary anymore. Installed package dependencies are saved directly to the project file instead.

9

Q9. What are strong-named assemblies and how do I create them?

Strong-named assemblies are digitally signed using a private strong name key. The signature is included in the assembly file along with the public key corresponding to the private key that was used for signing.

The mathematical concepts behind the [public key cryptography](#) ensure that anyone with the public key can verify the validity of the signature, i.e. make sure that the signed assembly hasn't changed since it was signed. To create a new signature for a modified file, access to the original private key is required.

Determining the private key solely based on its public key counterpart is computationally too demanding to make it feasible. This gives strong-named assemblies a unique identity which can be verified at runtime. Any assembly referencing a strong-named assembly also includes its public key token as a representation of the public key it was signed with. The verification is performed in two steps:

1. By checking the validity of the signature using the public key that's included in the assembly, to ensure that the assembly hasn't been tampered with since it was signed.
2. By checking the public key in the referenced assembly against the public key token stored in the referencing assembly, to ensure that the referenced assembly has been signed with the same private key.

A strong-named assembly cannot reference other assemblies that are not strong-named. This sometimes feels restrictive, especially when there's no strong-named version of a dependency we would like to use, but it also ensures that none of the dependencies has been tampered with. An assembly without a strong name could be recompiled from modified source code or even replaced altogether by a different assembly with the same name. Although it would still be a valid dependency for assemblies that reference it.

To create a modified but still valid strong-named assembly, the original private key is required, which should always be safeguarded by the owner. Albeit, strong-named assemblies do not provide any mechanisms of trust because they are signed with keys that are generated by developers themselves and don't include any information of ownership.

To provide the user with trustworthy information about the software publisher, Microsoft Authenticode signatures should be used instead. They can only be created with certificates issued by certification authorities which guarantee the identity of the prospective owner. Based on that assurance, Windows will show you information about the publisher when you try to install Authenticode signed software on your computer (See Figure 1). Strong-name signatures are not a replacement for this technology.

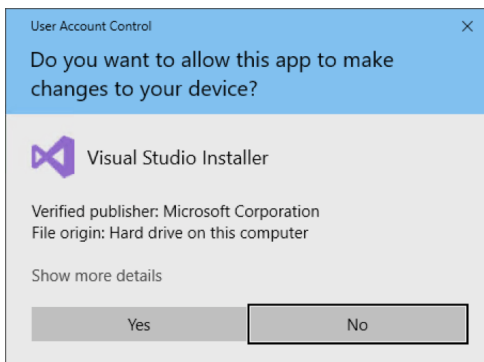


Figure 1: Dialog with publisher information for Authenticode signed software

To create your own strong-named assembly, you first need to generate a strong name key and then use it to sign the assembly after the build. The easiest way to achieve that is by using Visual Studio. The Signing tab of the project properties window contains all the required configuration settings:

- *Sign the assembly* checkbox
- *Choose a strong name key file* dropdown

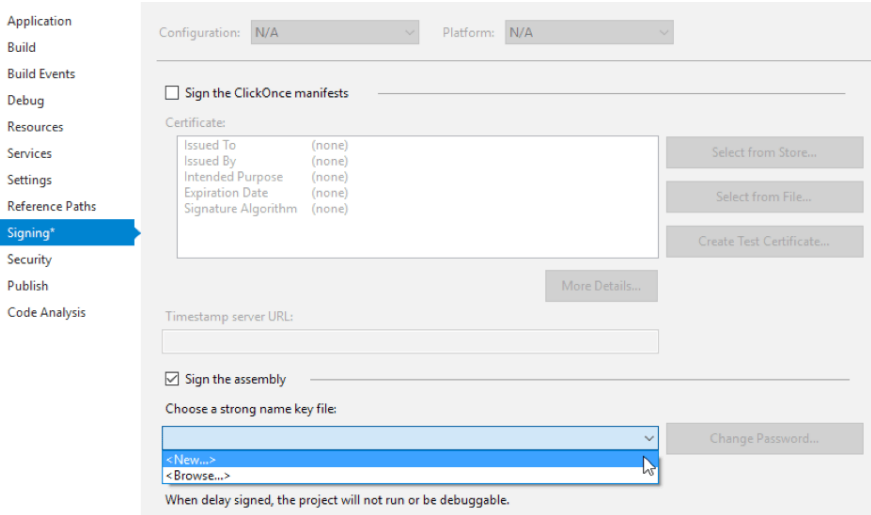


Figure 2: Signing tab in the project properties window

The dropdown will list all the key files which are already included in the project. On top of that there are two more options available:

- *Browse...* allows you to select a key file from elsewhere in the file system. It will automatically be copied to your project folder.
- *New...* opens a dialog which you can use to create a new key file if you don't have one yet.

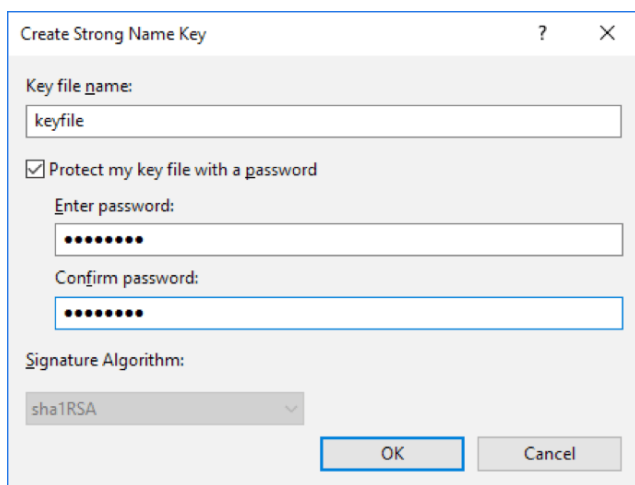


Figure 3: Dialog for creating a new strong name key

Alternatively, the key can be generated using the Sn.exe command line tool:

```
sn -k keyfile.snk
```

With strong name signing configured, the assembly will automatically be signed during the build process.

In larger organizations, it might be necessary to restrict access to the private strong name key so it isn't available to all developers. If devs still need their assemblies to behave in their development environment as if they were properly signed, they can use delay signing, which is exposed as an option on the *Signing* tab.

For delay signing, only the public key is sufficient. To extract one from the private key, Sn.exe can be used again:

```
sn -p keyfile.snk public.snk
```

In the development environment, signature verification can be *skipped* for such an assembly so that it can still be used, although it isn't yet signed properly. An assembly can be registered for verification skipping using Sn.exe:

```
sn -Vr DelaySignedAssembly.dll
```

Before the assembly is shipped into production it still needs to be properly signed with the private key. Usually this will happen as an additional step on the build server,

using Sn.exe of course:

```
sn -R DelaySignedAssembly.dll keyfile.snk
```

Strong-named assemblies can provide an additional challenge in open source scenarios. Although the source code of open source libraries is freely available and everyone can build their own (modified) copy of the assembly, the identity of this assembly would not match the original one if strong-name signing was used. To avoid that problem, the recommendation is to include the private key file in the repository together with the source code, so that everyone can sign the assemblies with it and still reference them from other strong-named assemblies.

.NET Core introduced the concept of public signing as an alternative to this approach.

It is very similar to delay signing in that it only requires the public key. However, the assemblies can still be used in most scenarios, even without registering them for signature verification skipping:

- In .NET Core, there are no restrictions at all, with the exception that public signed assemblies built from modified source code will not be supported by their original vendor.
- When used in the .NET framework, public-signed .NET Core assemblies (e.g. ASP.NET Core) can't be installed to the GAC (as explained later in the chapter).

With all .NET Core assemblies being open source, this allows painless recompilation of the code for everyone. It invalidates most of the originally claimed benefits of strong-named assemblies though. However, this is not a problem in .NET Core since strong-name signing is present only to provide backward compatibility with the .NET framework, and to allow servicing releases for .NET Core assemblies published by Microsoft.

Even in the .NET framework, the unique identification provided by strong-name signing is rarely useful for application developers. One such scenario is when assemblies need to be published to the Global Assembly Cache (GAC). Only strong-named assemblies can be installed into GAC.

If you have just started with .NET development, there's a possibility you haven't even heard of the GAC. There's nothing wrong with the concept since the latest official recommendation is that the GAC should only be used when it is necessary. There are only a couple of scenarios that require the GAC and they originate from the fact that

the GAC is a central install location for assemblies which need to be shared between multiple applications:

- Individual applications need not distribute and install their own copy of the dependent assembly and can thereby reduce the size of the distribution package and the install size. Of course, different applications might need different versions of dependent assemblies, therefore the GAC supports side-by-side installation of different versions of the same assembly.
- Servicing such shared assemblies when a security vulnerability is discovered and fixed becomes more manageable since there is only a single copy of that assembly in a known location, no matter how many applications depend on it. There's no need for each application to release an update with the fix. However, updating an assembly in the GAC usually requires administrator privileges because of the large impact caused by the change.

Assemblies are typically installed to the GAC using Windows Installer which includes a proper reference counting. This is done so that it can be determined when an assembly can safely be removed from the cache because no more applications depend on it. For administrative purposes, a command line utility (Gacutil.exe) is also available.

The storage location for the assemblies depends on the CLR version:

- Assemblies in CLR 2.0 GAC are stored in %windir%\assembly
- Assemblies in CLR 4 GAC are stored in %windir%\Microsoft.NET\assembly

However, the two directories should never be manipulated manually. Instead Gacutil.exe should always be used.

There is no GAC in .NET Core, nor any other concept resembling it. All assemblies except the ones distributed as part of a .NET Core release are always installed locally to the running application.

10

Q10. What should I know about Garbage Collection in .NET?

Garbage collection makes automatic memory management in .NET framework possible. Thanks to it, the developer is only responsible for allocating memory by creating new instances of objects. Allocated memory is automatically released by the garbage collector once the created objects are not used any more.

All managed objects in the .NET framework are allocated on the managed heap. They are placed contiguously as they are created. If this process would continue like that for a long time, we would eventually run out of memory. To prevent that, garbage collection gets triggered when memory allocations reach a certain total size or when there's a lack of available memory.

The garbage collection consists of three phases:

- In the **marking phase**, a list of all objects in use is created by following the references from all the root objects, i.e. variables on stack and static objects. Any allocated objects not on the list become candidates to be deleted from the heap. The objects on the list will be relocated in the compacting phase if necessary, to

compact the heap and remove any unused memory between them.

- In the **relocation phase**, all references to remaining objects are updated to point to the new location of objects to which they will be relocated in the next phase.
- In the **compacting phase**, the heap finally gets compacted as all the objects that are not in use any more are released and the remaining objects are moved accordingly. The order of remaining objects in the heap stays intact.

To better handle different lifetimes of different objects, garbage collection introduces the concept of **generations**. A newly created object is placed in generation 0. If it is not released during a garbage collection run, it is moved to generation 1. If it is still not released when the next garbage collection run happens, it is moved to generation 2. It stays there for the remainder of its lifetime.

Based on the assumption that most objects are only used for a short time, while the objects that are used longer are less likely to be released soon if at all, the generations are used to optimize the garbage collection process. Most garbage collection runs will only process generation 0 and will therefore take the shortest time. Once more memory needs to be released, generation 1 will be included in garbage collection as well, making it take more time. When even generation 1 garbage collection doesn't release enough memory, generation 2 gets included as well. This is called full garbage collection and takes the longest time.

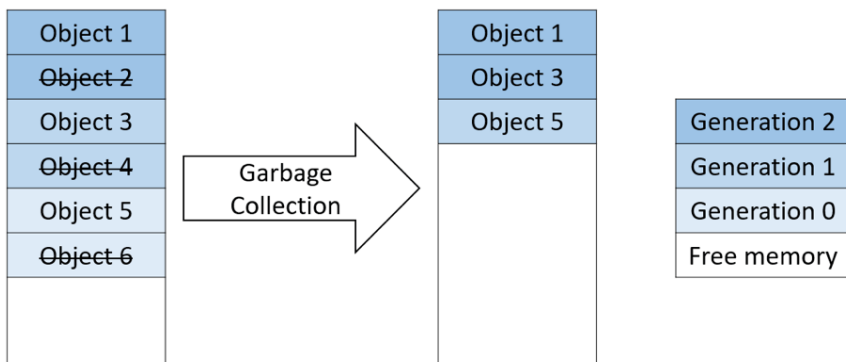


Figure 1: Garbage collection releases unused objects and promotes used objects through generations

To further optimize garbage collection, there is a separate object heap for objects larger than 85,000 bytes. Objects placed in it are handled differently:

- New objects are immediately put into generation 2, since larger objects (usually arrays) tend to live longer.
- Objects are not moved at the end of garbage collection to compact the large objects heap because moving larger objects would be more time consuming.

There are two different types of garbage collection:

- Workstation garbage collection runs on the normal priority thread, which triggered it by passing a memory threshold with its latest memory allocation or by explicitly calling `GC.Collect`
- Server garbage collection creates a separate managed heap and a corresponding garbage collection thread for each logical CPU. Threads run on highest priority. This approach to garbage collection makes it faster but more resource intensive.

By default, workstation garbage collection is used, but this can be changed with a setting in the application configuration XML file (named `AssemblyName.config`):

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <gcServer enabled="true"></gcServer>
  </runtime>
</configuration>
```

If the computer only has a single logical CPU, the setting will be ignored and the workstation garbage collection type will be used.

Each approach to garbage collection is available in three different variations:

- **Non-concurrent garbage collection** suspends all non-garbage-collection threads for the full duration of the garbage collection, effectively pausing the application for that time.
- **Concurrent garbage collection** allows user threads to run for the most of generation 2 garbage collection. As long as there is still free space in the managed heap for new allocations, user threads are allowed to run and create new objects. This results in a shorter garbage collection pause, at the cost of higher CPU and memory requirements.

- **Background garbage collection** is the replacement for concurrent garbage collection. It was introduced in .NET framework 4 for workstation garbage collection, and in .NET framework 4.5 for server garbage collection. It is similar to concurrent garbage collection but allows generation 0 and generation 1 garbage collection to interrupt an ongoing generation 2 garbage collection and temporarily block program execution. After generation 0 and generation 1 garbage collection is completed, both the program execution as well as the generation 2 garbage collection, continues. This even further shortens the time the program execution is paused because of garbage collection.

By default, concurrent or background garbage collection will be used (depending on the .NET framework version), but this can be changed with a setting in the application configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <gcConcurrent enabled="false"></gcConcurrent>
  </runtime>
</configuration>
```

Garbage Collection in .NET Core

The same two garbage collection configuration options are available .NET Core. Since there's no application configuration XML file in .NET Core, the settings are placed in the runtime configuration JSON file (named AssemblyName.runtimeconfig.json) instead:

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Concurrent": false,
      "System.GC.Server": true
    }
  }
}
```

Typically, this file is not edited manually. Its contents are generated at compile time based on the MSBuild project (.csproj) file. To configure the garbage collector, you can

set the following properties in the project file:

```
<PropertyGroup>  
  <ServerGarbageCollection>true</ServerGarbageCollection>  
  <ConcurrentGarbageCollection>>false</ConcurrentGarbageCollection>  
</PropertyGroup>
```

Garbage Collection - Best Practices

Taking all this knowledge into account, application performance can be improved by following these recommendations when garbage collection is identified as the cause for bad performance:

- When a multithreaded application creating a lot of objects is running on a computer which doesn't host many other processes, switch from workstation to server garbage collection. The application will benefit from shorter pauses while not negatively affecting other applications with high priority garbage collection threads.
- Although garbage collection can be triggered manually by calling `GC.Collect`, it should usually be avoided as the CLR heuristics will be able to better schedule garbage collection based on all the information available. Probably the only use case when manual garbage collection could make sense would be when in a short time, a lot of objects were not in use, and the application can afford a deterministic garbage collection pause without negatively affecting the user.
- Allocation of large objects with short life time should be avoided as much as possible. It will take a long time before they are released because they are immediately placed in generation 2 and they will make the large object heap fragmented because it is not compacted.
- If possible, avoid allocating large numbers of objects with short lifetimes and try to reuse the objects instead. This will avoid frequent triggering of generation 0 garbage collection, which causes pauses in program execution.

11

Q11. How are value and reference types different?

Types in the .NET framework can be grouped into two categories:

- **Value types:** all structs, including the built-in bool and numeric types (int, double and others)
- **Reference types:** all classes, including the built-in ones: string and object

To fully understand the differences between the two, we need to look at how data is stored in memory. Value types are usually stored on the stack, i.e. a linear memory structure best described with the following two rules:

- New values are always added at the end (the operation is usually called **push**).
- Values can only be removed from the end, i.e. in reverse order to how they were being added (the operation is usually called **pop**).

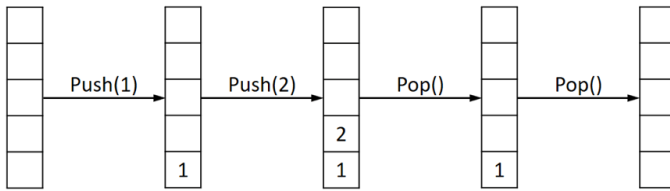


Figure 1: Stack data structure with push and pop operations

The values are only being added to the stack by the runtime as part of the method invocation process. When a method is called, the runtime creates a corresponding **stack frame**, containing all the values relevant to that invocation:

- the function parameters,
- the return address (used only internally – it indicates where the execution will continue after the function exits), and
- the local variables.

Every time a function is called, a new stack frame is pushed onto the stack, containing all its data. The stack frame is popped from the stack again when the function exits. Since the functions always exit in reverse order as they were called, a stack is the perfect data structure for storing their data.

All parameter and variable values are stored in-place, as part of the stack frame.

When a method is called and a local variable is passed as the function parameter, the value is copied. When the called function exits, the memory allocated for the copied values is released.

Let's look at how the stack contents change when the following code is run:

```
static void Method1()
{
    var a = 1;
    Method2(a);
}
static void Method2(int a)
{
    a = 2;
}
```

Parameters and local variables of each function live on the stack for the time from when the function was first called until its execution finally completes. When that happens, all the memory occupied by their values gets released immediately.

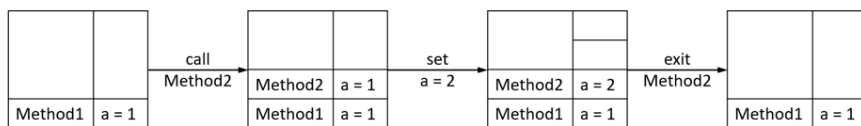


Figure 2: The stack contents during method execution

Although variables and parameters of reference types are also created on the stack, they don't directly contain their values. Instead, their values are stored on the **heap**, i.e. a tree data structure optimized for efficient dynamic allocation of memory. The value stored on the stack is only a reference pointing to the actual value in the heap. Memory on the heap is dynamically allocated whenever a new value is created using the **new** keyword. It is not directly tied to entering and exiting a function.

This affects the behavior when reference types are passed as parameter functions. Only a reference pointing at the value is copied and passed to the called function. After that, both references point at the same copy of the value in the heap.

When a new value is created using the *new* keyword and assigned to a variable, this value is newly allocated on the heap and the reference is changed to point to it. The old value gets eventually released by the garbage collector after no more references point to it.

Let's analyze this behavior with the following sample code:

```
static void Method1()
{
    var a = "1";
    Method2(a);
}

static void Method2(string a)
{
    a = "2";
}
```

Parameters and local variables still live only from when the function is called, until it completes. However, these variables don't contain the actual values which are stored

on the heap. When the references pointing to the values are removed together with their stack frame, the values still remain on the heap. Their memory will only be released during the next garbage collection if there are no other references pointing to them.

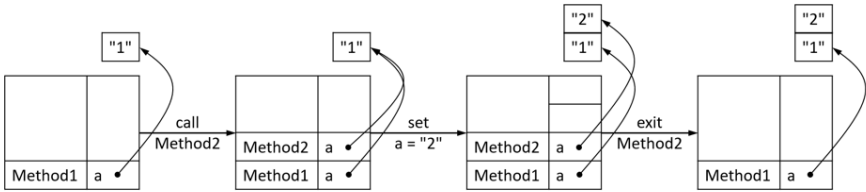


Figure 3: Stack and heap contents during method execution

These differences in memory management have their own advantages and disadvantages in different use cases:

- Value types are more suitable for smaller data types which are not much larger than the references themselves or are even smaller than they are (such as built-in numeric data types). Copying large blocks of values can take a long time and will increase memory consumption.
- In contrast, reference types are more suitable for larger data types where the size of the reference is only a small fraction of the full data type size (like strings for example). When passing them around, they will consume less memory because only references will be copied. However, avoid creating too many short-lived values when using reference types. Since they will not be automatically released when they are not used anymore, the memory consumption will temporarily increase. It will also affect performance as garbage collection will be called more often to release memory occupied by them.

12

Q12. What is the difference between classes and structs?

In many aspects classes and structs are very similar to each other. They are both composite data types which can contain any number of fields, properties, methods, and events.

Unlike classes, structs are always sealed and therefore cannot be inherited from. This also affects the access modifiers that can be used for its members – none of the protected modifiers can be used because they would only affect member accessibility in derived types.

The most important difference between the two is that classes are reference types while structs are value types. We are more used to the former as most data structures in the .NET framework class library are classes.

The values of reference types are stored on the heap. The stack only contains references pointing to these values. In the code, we are always passing around these references, therefore any changes we make to the value apply to all variables with a reference to that value.

The following sample code shows this behavior:

```
class Point
{
    public int x;
    public int y;
}
static void Method1()
{
    var a = new Point();
    Console.WriteLine($"Method1 - start: x = {a.x}, y = {a.y}");
    Method2(a);
    Console.WriteLine($"Method1 - end: x = {a.x}, y = {a.y}");
}
static void Method2(Point a)
{
    Console.WriteLine($"Method2 - start: x = {a.x}, y = {a.y}");
    a.x = 1;
    Console.WriteLine($"Method2 - end: x = {a.x}, y = {a.y}");
}
```

This is the console output after calling Method1:

```
Method1 - start: x = 0, y = 0
Method2 - start: x = 0, y = 0
Method2 - end: x = 1, y = 0
Method1 - end: x = 1, y = 0
```

The local variable of Method1 was passed to Method2. Method2 modified the common value in the heap, also affecting the variable value in Method1.

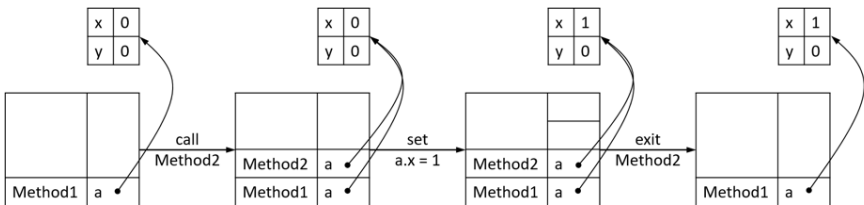


Figure 1: Variable values during method execution when using a class

Since a class is a reference type, it can also be set to **null** which means that it does not point to any value in the heap. It is treated as an empty or unknown value. Any

attempt of accessing one of its members will result in a *NullReferenceException*.

```
Point nullValue = null;
Point defaultValue = default(Point); // will also set value to null
var x = defaultValue.x; // will throw NullReferenceException
var y = defaultValue.y; // will throw NullReferenceException
```

Structs on the other hand are value types. This means that their value is usually stored directly on the stack.

There are two exceptions to that rule when structs can be stored on the heap:

- If a struct is a part of a reference type (e.g. a field in a class or inside an array)
- If a struct is boxed into a reference type (you can read more about boxing in the upcoming chapter 13 – "What is boxing and unboxing?")

When such a value is passed around, a copy is created. Any value changes to that copy do not affect the original value.

Let's look at a modified example from before:

```
struct Point
{
    public int x;
    public int y;
}
static void Method1()
{
    var a = new Point();
    Console.WriteLine($"Method1 - start: x = {a.x}, y = {a.y}");
    Method2(a);
    Console.WriteLine($"Method1 - end: x = {a.x}, y = {a.y}");
}
static void Method2(Point a)
{
    Console.WriteLine($"Method2 - start: x = {a.x}, y = {a.y}");
    a.x = 1;
    Console.WriteLine($"Method2 - end: x = {a.x}, y = {a.y}");
}
```

In this case the console output after calling Method 1 will be different:

```
Method1 - start: x = 0, y = 0
Method2 - start: x = 0, y = 0
Method2 - end: x = 1, y = 0
Method1 - end: x = 0, y = 0
```

The local variable of Method1 was passed to Method2. Although the parameter value was modified in Method2, the local variable value in Method1 remained unaffected.

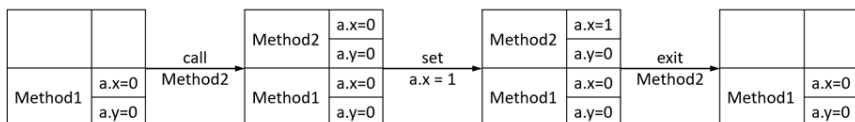


Figure 2: Variable values during method execution when using a struct

Such behavior is unexpected; therefore, it is good practice to always make structs immutable, i.e. prevent changing their fields after they are first created. This makes the above scenario impossible and forces the programmer to write the code differently:

```
struct Point
{
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; private set; }
    public int Y { get; private set; }
}

static void Method1()
{
    var a = new Point();
    Console.WriteLine($"Method1 - start: x = {a.X}, y = {a.Y}");
    a = Method2(a);
    Console.WriteLine($"Method1 - end: x = {a.X}, y = {a.Y}");
}
```

```
static Point Method2(Point a)
{
    Console.WriteLine($"Method2 - start: x = {a.X}, y = {a.Y}");
    a.X = 1; // will not compile
    a = new Point(1, a.Y);
    Console.WriteLine($"Method2 - end: x = {a.X}, y = {a.Y}");
    return a;
}
```

The console output will now be as expected:

```
Method1 - start: x = 0, y = 0
Method2 - start: x = 0, y = 0
Method2 - end: x = 1, y = 0
Method1 - end: x = 1, y = 0
```

Because a struct is a value type, its value cannot be set to `null`. A newly created struct will always have all its fields set to their default value:

```
Point nullValue = null; // will not compile
Point defaultValue = default(Point); // will set all fields to
default value
var x = defaultValue.X; // will set x to 0
var y = defaultValue.Y; // will set y to 0
```

Since structs can't have a custom default (parameterless) constructor defined, there's no way to change this default initialization. Invoking the parameterless constructor is equivalent to using the default expression for that struct.

```
Point defaultValue = new Point(); // equivalent to default
```

Of course, custom parameterized constructors are supported and can set the initial value of all fields to any value.

SECTION II

THE TYPE SYSTEM

13

Q13. What is boxing and unboxing?

Boxing is the process of casting a value type into a reference type.

```
int a = 42;
object o = a;
```

The cast is done implicitly when a value type is assigned to a variable of type *object*. With this operation, the value is copied from the stack to the heap so that it can be referenced from the new reference type variable.

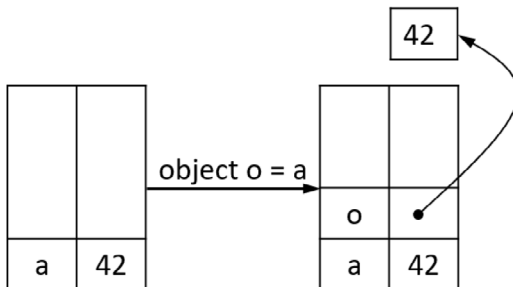


Figure 1: In-memory representation of boxing

Unboxing is the reverse operation of boxing, i.e. casting a boxed value back into the original value type:

```
object o = 42;
int a = (int)o;
```

Unlike boxing, unboxing requires an explicit cast. With this operation, the referenced value on the heap is copied into the stack.

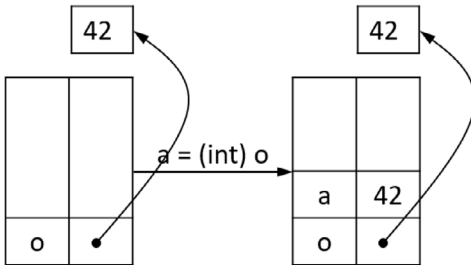


Figure 2: In-memory representation of unboxing

Unboxing is an operation which can throw exceptions at runtime:

- An attempt to unbox a null value will result in a `NullReferenceException` being thrown.
- Trying to unbox into a wrong type will throw an `InvalidCastException`.

Not only that. The cast used for unboxing must exactly match the original type or an `InvalidCastException` will still be thrown even if there is an implicit cast available between the boxed type, and the target type:

```
object o = 42;
long i = (int)o; // int will be implicitly cast to long
long l = (long)o; // InvalidCastException will be thrown
```

When a value type is boxed, a copy is created, therefore any modifications to the original value will not affect the boxed value.

```
struct Point
{
    public int X { get; set; }
```



```

    public int Y { get; set; }
}

static void Main(string[] args)
{
    Point p = new Point();
    object o = p;
    p.X = 1;
    Console.WriteLine($"p.X = {p.X}");
    Console.WriteLine($"o.X = {(Point)o.X}");
}

```

The output in this case would be:

```

p.X = 1
o.X = 0

```

Once the value is boxed, it can't be modified any more until it is unboxed. The modification of the unboxed value will not affect the boxed one because the value was copied in the process.

```

object o = new Point();
((Point)o).X = 1; // will not compile
Point p = (Point)o;
p.X = 1; // will leave the value of o unchanged

```

It's important to keep in mind that boxing and unboxing operations introduce a performance cost:

- Boxing a value type involves a new memory allocation on the heap, and copying of data. According to [the official documentation](#), it can take 20 times longer compared to a simple assignment of an existing reference type to another variable.
- According to [the official documentation](#), Unboxing requires only copying of data and has a smaller direct performance hit – up to 4 times longer compared to a simple reference assignment. However, the previously allocated memory on the heap for the boxed value will not be released on its own. It will require a garbage collection, which will cause additional indirect performance hit.

Because of that, **boxing and unboxing of a large number of values should be avoided, if possible.**

Boxing and Unboxing – Use cases

Let's look at the scenarios which might require the use of boxing. In general, boxing is a prerequisite for a unified type system in .NET.

This was even more important before generics, as it made it possible to use non-generic collections, such as `ArrayList` and `Hashtable` with value types:

```
var list = new ArrayList();
list.Add(1);
list.Add(2);
list.Add(3);
foreach (object i in list)
{
    Console.WriteLine(i);
}
```

Today, generic collections are a better, more performant alternative. They are designed to store value types directly and don't require any boxing or unboxing. This improves performance and preserves full type safety:

```
var list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
foreach (int i in list)
{
    Console.WriteLine(i);
}
```

Nevertheless, there are still scenarios which might require the use of boxing. For example, a collection of mixed value types still needs to treat all of them as objects:

```
var list = new List<object>();
list.Add(42);
list.Add(true);
list.Add(3.14);
```

```
foreach (object i in list)
{
    Console.WriteLine(i);
}
```

Fortunately, this is not a common use case. Still, in some cases you cannot avoid boxing because of the way the APIs are designed:

```
String.Format("{0} = {1}", "boxing", true);
```

In this method call, the second and the third argument are typed as object. Therefore, the bool value will be boxed before it is passed to the function (string is already a reference type and can be cast to object without boxing).

Such cases are not too common in the .NET framework class library and it is quite unlikely that using these APIs would result in excessive boxing. It is something to keep in mind when designing your own APIs, though. You should always prefer generic argument types if they can be used instead.

Boxing is also involved when a struct is cast to an interface it implements, because interfaces can also be implemented by classes and are therefore universally treated as reference types.

```
interface IPoint
{
    int X { get; set; }
    int Y { get; set; }
}

struct Point : IPoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

static void Main(string[] args)
{
    Point p = new Point();
    IPoint i1 = p;
```

```
IPoint i2 = i1;  
i1.X = 1;  
Console.WriteLine($"p.X = {p.X}");  
Console.WriteLine($"i1.X = {i1.X}");  
Console.WriteLine($"i2.X = {i2.X}");  
}
```

This code will give the following output:

```
p.X = 0  
i1.X = 1  
i2.X = 1
```

The value of the original point is copied only when it is first cast to the interface. Because the interface is treated as a reference type, the cast value is stored on the heap from there on. When it is assigned to the second variable of the interface type, only a reference is assigned to it. Any modifications of the interface value are now common to both references pointing to it but don't affect the original copy of the struct in the stack.

If interfaces were treated as value types, all the values would be stored on the stack and each of the three variables would hold its own copy. Modifying one interface value would then not affect the other one.

14

Q14. Why are nullable types applicable only to value types?

There's a fundamental difference between value types and reference types which affects the ability of representing an empty value:

- Reference types contain a reference pointing to the actual value stored in the memory. When this reference is set to `null`, it means that it does not point to any memory location and the value is therefore empty.
- Value types directly contain the value. Therefore, whatever value they are set to, it is the actual value, which makes it impossible to represent an empty value.

.NET framework 2.0 introduced the generic `Nullable<T>` struct which can be used to represent an empty/missing value for a value type. There's no need to use it with reference types because they can already represent the empty/missing value on their own. Therefore, the generic argument type in `Nullable<T>` is also constrained to value types. If we accidentally tried to put a reference value into a `Nullable<T>`, it would result in a compiler error.

At its core, `Nullable<T>` wraps a value type and adds an extra `HasValue` property to

indicate whether the value is set or not:

```
var nonEmpty = new Nullable<int>(0);
Console.WriteLine(nonEmpty.HasValue);
var empty = new Nullable<int>();
Console.WriteLine(empty.HasValue);
```

The above code will give the following output as expected:

```
True
False
```

When the value is set, the Value property can be used to access it:

```
var nonEmpty = new Nullable<int>(0);
Console.WriteLine(nonEmpty.Value);
var empty = new Nullable<int>();
Console.WriteLine(empty.HasValue);
```

When the value is not set, accessing the Value property will throw an exception of type `InvalidOperationException`.

There is some syntactic sugar available to make the usage more convenient:

- `Nullable<T>` has a shorter notation: `T?`

```
var nonEmpty = new int?(0);
var empty = new int?();
```

- There is an implicit cast available from null and the original value type, to its `Nullable<T>` representation:

```
int? nonEmpty = 0;
int? empty = null;
```

- There is an explicit cast available from `Nullable<T>` to the corresponding value type:

```
int a = (int)nonEmpty;
```

The explicit cast is the equivalent of accessing the Value property. If no value is set, it will also throw an exception of type `InvalidOperationException`.

Boxing and unboxing operations also have customized behavior for `Nullable<T>`. When boxing a `Nullable<T>`, the entire `Nullable<T>` struct does not get copied to the heap. Instead the behavior is different based on whether the value is set or not:

- When the value is set, only the value gets copied to the heap, just like when boxing the value type itself. The boxed value can then be unboxed either back to a `Nullable<T>` or directly to the underlying value type.

```
int? nonEmpty = 0;
var boxed = (object)nonEmpty;
int? unboxedNullable = (int?)boxed;
int unboxedInt = (int)boxed;
```

- When the value is not set, no value gets copied to the heap. The reference is simply set to `null`. The value can still be unboxed back to `Nullable<T>`. Trying to unbox the resulting `null` value to the underlying value type will result in an exception of type `NullReferenceException` being thrown.

```
int? empty = null;
var boxed = (object)empty;
int? unboxedNullable = (int?)boxed;
```

All of this makes the handling of `null` values when using nullable types, very similar to reference types. Most of the time we can just forget that we're dealing with value types, which inherently are not nullable.

15

Q15. How are enumeration types supported in C#?

Enumeration types can be declared to assign meaningful names to selected integer values. These can then be used elsewhere in code instead of [magic numbers](#), making the code easier to understand and maintain. For example, instead of using an ordinal number for each day of the week, we can use their names:

```
public enum DayOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

if (currentDay == 7) // Sunday as a magic number
```



```

{
    // ...
}

if (currentDay == DayOfWeek.Sunday) // Sunday as a day with a name
{
    // ...
}

```

If we don't care which values are used to represent the declared names, we can omit the values in the declaration:

```

public enum DayOfWeek
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

```

In this case, sequential numbers will be used, starting with 0. We can even set a value for only some of the names in the enumeration type. Names that don't have their value set will simply use the value of the previous name incremented by 1. By explicitly setting a value, multiple names can use the same value:

```

public enum Quantity
{
    None,
    Few = 5,
    Six,
    Many = 100,
    Lots = 100
}

```

By default, the underlying type for enumeration types is `int`, but any other integral type (such as `byte` or `long`) can be used instead:

```
public enum LargeValues : long
{
    TenBillions = 10_000_000_000
}
```

There are cast operators available to convert a value between the enumeration type and its underlying type. Usually, they can be avoided unless the code is interacting with other systems depending on the numeric value, e.g. databases. Casting from a numeric value back to an enumeration type can be particularly dangerous, as there is no check performed, whether there is a name assigned to that value or not.

```
int asNumeric = (int)DayOfWeek.Sunday;
DayOfWeek asEnumeration = (DayOfWeek)100; // no exception will be
thrown
```

C# provides additional functionality for enumeration types which are meant to be used as bit flags:

```
[Flags]
public enum DayOfWeek
{
    None = 0x0,
    Monday = 0x1,
    Tuesday = 0x2,
    Wednesday = 0x4,
    Thursday = 0x8,
    Friday = 0x10,
    Saturday = 0x20,
    Sunday = 0x40
}
```

In this example, I'm using hexadecimal literals instead of the usual decimal literals in order to make their bit representation more evident.

Since C# 7, binary literals might be an even better choice:

```
[Flags]
public enum DayOfWeek
{
```

```

None = 0x0,
Monday = 0b1,
Tuesday = 0b10,
Wednesday = 0b100,
Thursday = 0b1000,
Friday = 0b1_0000,
Saturday = 0b10_0000,
Sunday = 0b100_0000
}

```

Each value in the above enumeration type sets a different bit in its binary representation. `DayOfWeek.None` is a special case as it represents a value with no bits set. It's a good practice to always include it in such enumeration types.

When each value in an enumeration type only has a single bit set in its binary representation, then these values can be combined and checked for:

```

var weekend = DayOfWeek.Saturday | DayOfWeek.Sunday; // =
0b110_0000
var isDayInWeekend = (weekend & day) == day;

```

Include common combinations of values directly in the enumeration type to make them more convenient to use:

```

[Flags]
public enum DayOfWeek
{
    None = 0x0,
    Monday = 0b1,
    Tuesday = 0b10,
    Wednesday = 0b100,
    Thursday = 0b1000,
    Friday = 0b1_0000,
    Saturday = 0b10_0000,
    Sunday = 0b100_0000,
    Weekday = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekend = Saturday | Sunday
}

```

In .NET 4 and later versions, there's a better way to check whether a flag has been set. Instead of using the binary "and" operator "&", the **HasFlag** method can be used:

```
var isInWeekend = DayOfWeek.Weekend.HasFlag(day);
```

The **FlagsAttribute** that I used on the enumeration type doesn't affect any of the code so far. It does change the behavior of the **ToString** method though:

```
var firstTwoDays = DayOfWeek.Monday | DayOfWeek.Tuesday;  
Console.WriteLine(firstTwoDays.ToString());
```

With the **FlagsAttribute** set, the code would output the following to the console:

```
Monday, Tuesday
```

Without the attribute, the output would simply be: 3

However, the attribute does not affect the reverse method **Enum.TryParse**:

```
DayOfWeek parsed;  
Enum.TryParse("Monday, Tuesday", out parsed);
```

Even without the attribute set, this code would correctly parse the string representation of the flags and set the parsed variable value accordingly.

As you can see, there's only a minor difference in behavior of enumeration types with and without the **FlagsAttribute** set. Despite that, **it's a good practice to always set this attribute when the values in the enumeration type are meant to be used as flags**. It will clearly document the intention for anyone reading or modifying the code at a later time.

16

Q16. How can multi-dimensional data be modelled using arrays?

In their simplest form, arrays are a single-dimensional ordered collection of values of the same type which can be accessed by their zero-based indexes. Their size is set when they are created; either uninitialized, or with an initial collection of values:

```
var uninitializedArray = new int[10];  
var initializedArray = new[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
var valueFromArray = initializedArray[0];
```

Arrays are always allocated as a contiguous block of memory for all the values in the array. For value types, the values are stored directly in the array. For reference types, the array contains only references to the objects allocated on the heap.

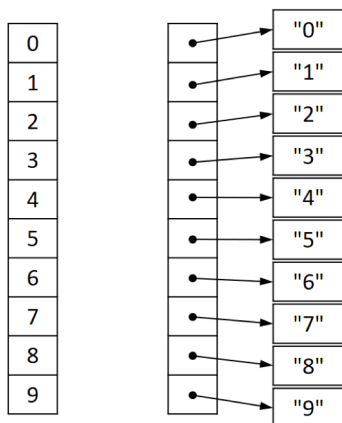


Figure 1: An array of value types vs. an array of reference types

Their size can be inspected at runtime using the `Length` property. To change their size, `Array.Resize` can be used. Use it with caution when dealing with large arrays, because it needs to allocate a completely new array of the requested size, and then copy across all the values from the old array.

```
Array.Resize(ref initializedArray, initializedArray.Length + 1);
```

To model multi-dimensional data with arrays, two different approaches can be used. The obvious one relies on multi-dimensional arrays. They are like single-dimensional arrays, except that their size is specified for more than one dimension.

```
var uninitialized2dArray = new int[5, 5];
var initialized2dArray = new[,]
{
    { 0, 1, 2, 3, 4 },
    { 1, 2, 3, 4, 5 },
    { 2, 3, 4, 5, 6 },
    { 3, 4, 5, 6, 7 },
    { 4, 5, 6, 7, 8 }
};
var valueFrom2dArray = initialized2dArray[0, 0];
```

We can imagine that in a two-dimensional array, the values are organized in a table instead of in a single row or list.

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

0	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

Figure 2: A single-dimensional array and a two-dimensional array

A multi-dimensional array still needs to be allocated as a contiguous block of memory. Based on index values, the offset from the starting memory location can be calculated for each value.

The value returned by the **Length** property of a multi-dimensional array might seem a bit surprising – it returns the total number of elements in the array across all dimensions. To get the size for individual dimensions, the **Rank** property and the **GetLength** method must be used instead:

```
var totalLength = initialized2dArray.Length; // = 25
var dimensionsCount = initialized2dArray.Rank; // = 2
var firstDimensionLength = initialized2dArray.GetLength(0); // = 5
var secondDimensionLength = initialized2dArray.GetLength(1); // = 5
```

There's no built-in **Array.Resize** method equivalent available for multi-dimensional arrays. You could implement it yourself by creating a new multi-dimensional array with the desired size and then correctly copying the values across for each dimension. However, the end result will be even more inefficient than in the case of a single-dimensional array.

The alternative approach to modelling multi-dimensional data uses arrays of arrays, also called jagged arrays. In the two-dimensional case, the first dimension would be a regular single-dimensional array, containing single-dimensional arrays of values, instead of the values directly:

```
var uninitializedJaggedArray = new[]
{
    new int[5],
```

```

    new int[5],
    new int[5],
    new int[5],
    new int[5]
};

var initializedJaggedArray = new int[][]
{
    new[] { 0, 1, 2, 3, 4 },
    new[] { 1, 2, 3, 4, 5 },
    new[] { 2, 3, 4, 5, 6 },
    new[] { 3, 4, 5, 6, 7 },
    new[] { 4, 5, 6, 7, 8 }
};

var valueFromJaggedArray = initializedJaggedArray[0][0];

```

Because of different structuring of data, the syntax for accessing individual values is also different. Instead of listing all the indexes in a single pair of brackets, each index needs to be inside its own brackets.

In a two-dimensional jagged array, the first index selects the inner array and the second index selects a value inside that array.

The memory for jagged arrays is not allocated contiguously any more. Each array (the root array for the first dimension and each of the child ones for the second dimension) is allocated independently of the others in a separate block of memory:

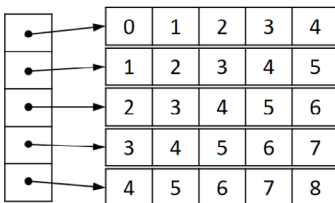


Figure 3: Memory allocation for a jagged array

This also allows individual inner arrays to be of different lengths. That's where the

name jagged comes from.

```
var jaggedArray = new int[][]
{
    new[] { 0 },
    new[] { 1, 2 },
    new[] { 1, 2, 3 },
    new[] { 1, 2, 3, 4 },
    new[] { 1, 2, 3, 4, 5 }
};
```

While such a jagged data structure can be useful and more effective in certain scenarios, it requires additional caution when accessing individual values. The irregular structure of data increases the risk of attempting to access a non-existent value with an out-of-range index.

```
var nonExistentValue = jaggedArray[0][2]; // throws
IndexOutOfRangeException
```

Since a jagged array is not a single data structure, but a combination of multiple ones, the **Length** and **Rank** properties behave accordingly.

```
var rootRank = jaggedArray.Rank; // = 1
var rootLength = jaggedArray.Length; // = 5
var inner0Rank = jaggedArray[0].Rank; // = 1
var inner0Length = jaggedArray[0].Length; // = 1
var inner1Rank = jaggedArray[1].Rank; // = 1
var inner1Length = jaggedArray[1].Length; // = 2
```

The **Resize** method works as expected for each individual array as well.

17

Q17. What are some of the less commonly used C# operators?

There's a large collection of operators available in C#. Many are common to most programming languages and we use them regularly, e.g. arithmetic, relational, logical, and equality operators. Others might not be as generally useful, but can make your code simpler and more efficient in specific scenarios.

The conditional operator is a great replacement for simple if-else statements used only to assign one or the other value to the same variable.

```
if (compactMode)
{
    itemsPerRow = 5;
}
else
{
    itemsPerRow = 3;
}
```

The same result can be achieved using a single expression with the conditional operator.

```
itemsPerRow = compactMode ? 5 : 3;
```

This operator is especially convenient when used with simple expressions. Checking if a value is null would be another good example:

```
middleName = middleName != null ? middleName : "";
```

In this special case, the null-coalescing operator can be used instead.

```
middleName = middleName ?? "";
```

Unfortunately, it only works when we want to directly use the value being checked for null. If we want to access one of its members, we still need the conditional operator.

```
count = list != null ? list.Count : 0;
```

To simplify such cases, the null conditional operator was introduced in C# 6.

```
count = list?.Count ?? 0;
```

It allows safe member access. If the left-hand side value is non-null, it accesses the member. Otherwise it just returns `null` avoiding the `NullReferenceException` which would be thrown when trying to access a member of a null-valued variable.

Its usage is not restricted to properties. It also works with fields, methods and even index access.

```
int? firstValue = list?[0];
```

The null conditional operator is also a great choice for thread-safe invoking of event handlers.

In earlier versions of C#, the event handler had to be stored in a local variable to be safe in multi-threaded scenarios. Otherwise its value could be changed by another thread after it was checked for null but before it was invoked, causing a `NullReferenceException`.

```
public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged(string propertyName)
{
    var handler = PropertyChanged;
    if (handler != null)
    {
        handler.Invoke(this, new
            PropertyChangedEventArgs(propertyName));
    }
}
```

This whole block of code can be replaced with a single line if the null conditional operator is used:

```
public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new
        PropertyChangedEventArgs(propertyName));
}
```

The call is still thread-safe, i.e. the generated code evaluates the variable only once. It then keeps the result in a temporary variable so that it cannot be changed afterwards.

Bitwise operators are specialized operators for bit manipulation of 32-bit (int and uint) and 64-bit (long and ulong) numeric values. Let's take advantage of binary literals and digit separators (both were added in C# 7.0) to see how they behave:

```
int a = 0b00000000_00000000_00000000_00001111;
int b = 0b00000000_00000000_00000000_01010101;
```

The result of the bitwise AND operator will include the bits which are set in both operands.

```
var and = a & b; // = 0b00000000_00000000_00000000_00000101
```

The result of the bitwise OR operator will include the bits which are set in at least one operand.

```
var or = a | b; // = 0b00000000_00000000_00000000_01011111
```

The result of the bitwise XOR (exclusive or) operator will include the bits which are set in exactly one operand.

```
var xor = a ^ b; // = 0b00000000_00000000_00000000_01011010
```

The result of the bitwise complement operator will include the bits which are not set in its (single) operand.

```
var complement = ~a; // = 0b11111111_11111111_11111111_11110000
```

The bitwise shift operators shift the bits in the binary representation to the left or to the right by the given number of places.

```
var shiftLeft = a << 1; // = 0b00000000_00000000_00000000_00011110
var shiftRight = a >> 1; // = 0b00000000_00000000_00000000_00000111
```

The bits which move past the end, do not wrap around to the other side. Hence, if we shifted any value far enough to the left or to the right, the result will be 0.

```
var multiShift = 0b00000000_00000000_00000000_00000001;
for (int i = 0; i < 32; i++)
{
    multiShift = multiShift << 1;
}
```

However, if we use the second operand to shift the bits by the same number of places in a single step, the result won't be the same.

```
var singleShift = 0b1 << 32; // = 0b1
```

To understand this behavior, we must look at how the operator is defined.

Before shifting the bits, the second operand will be normalized to the bit length of the first operand with the modulo operation, i.e. by calculating the remainder of dividing the second operand by the bit length of the first operand. In our example, the first operand is a 32-bit number, hence $32 \% 32 = 0$. The number will be shifted left by 0 places, not 32.

The bitwise AND, OR and XOR operators also serve as **logical operators** when applied to bool operands.

```
var a = true;
var b = false;
var and = a & b; // = false
var or = a | b; // = true
var xor = a ^ b; // = true
```

Although these AND and OR operators return the same result as the corresponding **conditional operators** (&& and ||), they don't behave identically. Operators & and | are eager, i.e. they will evaluate both operands even if the result can be determined after evaluating the first operand. Operators && and || are lazy, i.e. they won't evaluate the second operand when the result can be determined based on the value of the first operand. This can be an important difference when operands are method calls instead of simple variables:

```
var eager = true | IsOdd(1); // IsOdd will be invoked
var lazy = true || IsOdd(1); // IsOdd won't be invoked
```

All arithmetic and bitwise operators also have a corresponding shorthand assignment operator for the case when the calculated value is assigned to the first operand.

```
a = a + b; // basic syntax
a += b; // shorthand syntax
```

A special case of arithmetic operators is increment and decrement operators, which respectively increment or decrement the operand value by 1. They are available in two flavors: prefix and postfix. As the name implies, the former changes the operand before returning its value, while the latter first returns the operand value, and then changes it.

```
var n = 1;
var prefixIncrement = ++n; // = 2, n = 2
var prefixDecrement = --n; // = 1, n = 1
var postfixIncrement = n++; // = 1, n = 2
var postfixDecrement = n--; // = 2, n = 1
```

All four operators are most commonly used in **for** loops to change the value of the loop variable.

18

Q18. How can method parameters be passed by reference?

Method parameters in C# are passed by value unless explicitly specified otherwise. This means that a copy of the parameter is passed into the method. Depending on whether the parameter is value-typed or reference-typed, the resulting behavior is different because of how value types and reference types are represented in memory.

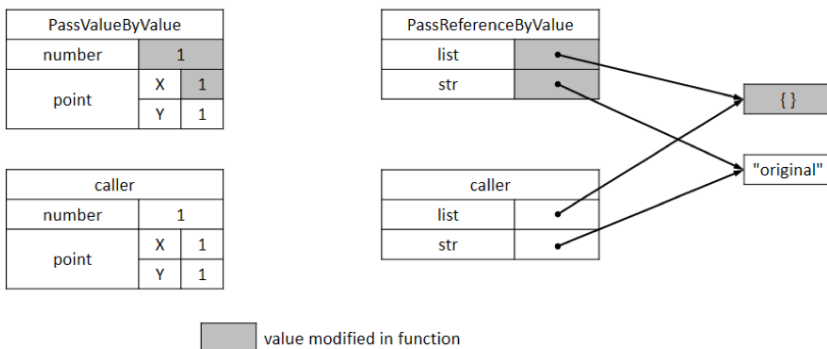


Figure 1: Passing value and reference types by value

For value types (either a primitive type or a structure), the actual value is copied into the method. This means that any changes to the local copy will not affect the variable outside the method.

```
void PassValueByValue(int number, Point point)
{
    number += 1;
    point.X += 1;
}

var number = 1;
var point = new Point(1, 1); // Point is a struct
PassValueByValue(number, point);
// number = 1
// point.X = 1, point.Y = 1
```

For reference types, the reference to the actual value is copied into the method. This means that any changes to the object passed into the method will also be visible outside the method. However, assigning a different value to the parameter directly won't affect the external variable.

The latter is particularly relevant to strings which are immutable, i.e. their value can't be changed after they are created. Because of that, the value of a string variable which is passed as an argument into a method can't be modified from inside that method, even though the string is a reference type. If a new value is assigned to the string parameter, that value will not be visible outside the method.

```
void PassReferenceByValue(List<int> list, string str)
{
    list.Add(1);
    list = new List<int>(new[] { 2 });
    str = "modified";
}

var list = new List<int>();
var str = "original";
PassReferenceByValue(list, str);
// list = { 1 }
// str = "original"
```


Alternatively, parameters can be passed into a method by reference when preceded by the **ref** keyword. This means that a reference to the parameter is passed into the method. Therefore, both value-typed and reference-typed parameters can be completely changed inside the method.

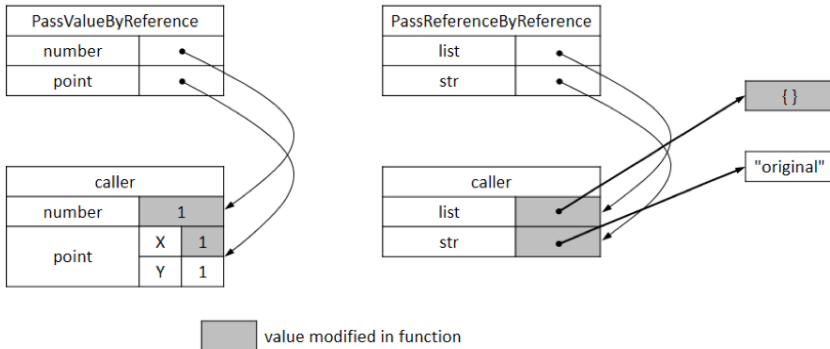


Figure 2: Passing value and reference types by reference

For value types, a reference to the *actual value* is passed.

```
void PassValueByReference(ref int number, ref Point point)
{
    number += 1;
    point.X += 1;
}
```

```
var number = 1;
var point = new Point(1, 1);
PassValueByReference(ref number, ref point);
// number = 2
// point.X = 2, point.Y = 1
```

For reference types, a reference to the *reference to the actual value* is passed.

```
void PassReferenceByReference(ref List<int> list, ref string str) {
    list.Add(1);
    list = new List<int>(new[] { 2 });
    str = "modified";
}
```

```
var list = new List<int>();
var str = "original";
PassReferenceByReference(ref list, ref str);
// list = { 2 }
// str = "modified"
```

The **out** keyword is used for passing parameters by reference in a slightly different way. As the meaning of the keyword implies, no value is passed into the method. Out parameters are only used to output values from the method. This means that an out parameter (either value-typed or reference-typed) must be initialized inside the method before the method exits. Even the method itself isn't allowed to read the out parameter value before initializing it.

For value types, a reference to the actual value is output to the caller and overwrites any value the variable might have had.

```
void PassValueAsOut(out int number, out Point point)
{
    number += 1; // will not compile
    number = 2;
    point.X += 1; // will not compile
    point = new Point(2, 1);
}
var number = 1;
var point = new Point(1, 1);
PassValueAsOut(out number, out point);
// number = 2
// point.X = 2, point.Y = 1
```

For reference types, a reference to the reference to the actual value is output to the caller and overwrites any value the variable might have had before.

```
void PassReferenceAsOut(out List<int> list, out string str)
{
    list.Add(1); // will not compile
    list = new List<int>(new[] { 2 });
    str = "modified";
}
```

```
var list = new List<int>();
var str = "original";
PassReferenceAsOut(out list, out str);
// list = { 2 }
// str = "modified"
```

In C#, a method can be overloaded (i.e. two methods with the same name can exist) based on whether parameters are passed by value or by reference.

```
void OverloadedMethod(int number)
{
    // method implementation
}

void OverloadedMethod(ref int number)
{
    // method implementation
}
```

However, overloading is not possible based only on the distinction between **ref** and **out** type parameters passed by reference.

```
void OverloadedMethod(ref int number)
{
    // method implementation
}

// will not compile
void OverloadedMethod(out int number)
{
    // method implementation
}
```

For the code to build successfully, one of the two options must be chosen, or the methods need to be named differently.

19

Q19. When does using anonymous types make sense?

Anonymous types are created implicitly when an object initializer is used **without specifying the type**.

```
var person = new
{
    FirstName = "John",
    LastName = "Doe"
};
```

The compiler will automatically generate a corresponding type under the hood. All properties listed in the initialization statement will be created as read-only properties. No other members will be added to the type.

Thanks to sensible overrides for the `Equals` and `GetHashCode` methods, equality is not reference based. Instead, two instances of the generated type are equal when all their properties are equal. There's also an override for the `ToString` method which returns nicely formatted values of all the properties.

```
Console.WriteLine(person.ToString());
// Output: { FirstName = John, LastName = Doe }
```

To create a class with semantics similar to the generated one, you could write the following code:

```
internal sealed class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public override bool Equals(object obj)
    {
        var value = obj as Person;
        if (value == null)
        {
            return false;
        }

        return Equals(FirstName, value.FirstName) && Equals(LastName,
            value.LastName);
    }

    public override int GetHashCode()
    {
        return FirstName?.GetHashCode() ?? 0 ^ LastName?.GetHashCode()
            ?? 0;
    }

    public override string ToString()
    {
        return $"{FirstName = {FirstName}, LastName = {LastName} }";
    }
}
```

You might have noticed that this class cannot be used with an object initializer because it doesn't have a default parameterless constructor, and the properties don't have a public setter. However, that's how the anonymous type is generated as well. Since object initializers can't be used for a type with read-only properties, the compiler converts the initializer syntax to a constructor call.

Nevertheless, this code isn't identical to the one generated by the compiler. It will behave similar enough though.

Note: *If you're curious how the generated code really looks like, you can create a simple application which uses an anonymous type and open the compiled assembly in one of the .NET decompilers. It will show you an approximation of the code which can be used to generate an equivalent type yourself.*

When an anonymous type is created by mapping properties from another type, it's not necessary to explicitly set the name for each property of the resulting anonymous type. If it's not set, the property name from the source type will be used, assuming this will ensure unique names for all properties in the anonymous type.

```
var aggregatedPerson = new
{
    person.FirstName,
    person.LastName,
    person.Address.Country,
    FatherName = person.Father.FirstName
};
```

In this code, we only had to set the name for the last property because the source property name was the same as that of the first property (FirstName). The code will initialize exactly the same type as in the example below where we explicitly set a name for each property.

```
var aggregatedPerson = new
{
    FirstName = person.FirstName,
    LastName = person.LastName,
    Country = person.Address.Country,
    FatherName = person.Father.FirstName
};
```

The main advantage of anonymous types is that we can use them without explicitly declaring them. As we have seen in one of the previous examples, there's quite some code required in order to achieve the same behavior. Within a single assembly, the compiler will use the same generated type for all instances which have properties of the same name and type generated in the same order. This allows you to declare multiple variables of the same anonymous type and then process them together, e.g. after adding both of them to the same array:

```
var person1 = new
{
    FirstName = "Anders",
    LastName = "Hejlsberg"
};
var person2 = new
{
    FirstName = "Mads",
    LastName = "Torgersen"
};
var persons = new[]
{
    person1,
    person2
};
foreach (var person in persons)
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

The name of the type is visible only to the compiler and isn't accessible from the code. Hence, the type can only be used with implicitly typed local variables (declared with the `var` keyword), but not anywhere else, e.g. for fields, properties, parameters, return values, etc. This limitation can be circumvented by declaring a dynamically typed member using the `dynamic` keyword. However, that would defeat the purpose since we would lose all the benefits of strong typing.

With all this in mind, there's only a limited set of scenarios where anonymous types are really useful. The most common case are ad-hoc types which only need to be used inside a single function. Not having to create custom inner types for them can noticeably reduce the amount of code that needs to be written.

This description fits the context of LINQ (Language INtegrated Query). It shouldn't be a surprise that anonymous types were introduced to C# as part of the .NET framework 3.5 together with LINQ. They are most commonly used inside the `select` clause to project only a subset of properties from the source type into the target anonymous type.

```
var personNames =  
    from p in persons  
    select new { p.FirstName, p.LastName };
```

Certain LINQ providers take advantage of this to only retrieve the requested subset of values from the data store, instead of all of them. A good example is LINQ to Entities which generates appropriate SQL queries based on the properties in the [Entity Framework](#) model classes and their subset included in the target anonymous type.

20

Q20. What does it mean that C# is a type safe language?

Type safety is a principle in programming languages which prevents accessing of variable values assigned inconsistently with their declared type. There are two kinds of type safety based on when the checks are performed:

- Static type safety is checked at compile time.
- Dynamic type safety is checked at runtime.

Without the checks, the values could be read from the memory as if they were of another type than they are, which would result in undefined behavior and data corruption.

C# implements measures for both kinds of type safety.

Static type safety prevents access to non-existent members of the declared type:

```
string text = "String value";  
int textLength = text.Length;
```

```
int textMonth = text.Month; // won't compile
```

```
DateTime date = DateTime.Now;  
int dateLength = date.Length; // won't compile  
int dateMonth = date.Month;
```

The two lines marked with a comment won't compile, because the declared type of each variable doesn't have the member we are trying to access. This check can't be circumvented even if we try to cast the variable to the type which does have the requested member:

```
string text = "String value";  
int textLength = text.Length;  
int textMonth = ((DateTime)text).Month; // won't compile
```

```
DateTime date = DateTime.Now;  
int dateLength = ((string)date).Length; // won't compile  
int dateMonth = date.Month;
```

The two lines with comments still won't compile. Although the type we're trying to cast the value to has the member we're accessing, the compiler doesn't allow the cast because there is no cast operation defined for converting between the two types.

However, static type safety in C# is not 100% reliable. By using specific language constructs, the static type checks can still be circumvented, as demonstrated by the following code:

```
public interface IGeometricShape  
{  
    double Circumference { get; }  
    double Area { get; }  
}  
  
public class Square : IGeometricShape  
{  
    public double Side { get; set; }  
    public double Circumference => 4 * Side;  
    public double Area => Side * Side;  
}
```

```

public class Circle : IGeometricShape
{
    public double Radius { get; set; }
    public double Circumference => 2 * Math.PI * Radius;
    public double Area => Math.PI * Radius * Radius;
}

IGeometricShape circle = new Circle { Radius = 1 };
Square square = ((Square)circle); // no compiler error
var side = square.Side;

```

The line marked with the comment will compile without errors, although we can see from the code that the `circle` variable contains a value of type `Circle` which can't be cast to the `Square` type. The compiler does not perform the static analysis necessary to determine that the `circle` variable will always contain a value of type `Circle`. It only checks the type of the variable. Since the compiler allows downcasting from the base type (the `IGeometricShape` interface) to a derived type (the `Square` type) as it might be valid for certain values of the variable, our code will compile.

Despite that, no data corruption will happen because of this invalid cast. At runtime, the compiled code will not execute. It will throw an `InvalidCastException` when it detects that the value in the `circle` variable can't be cast to the `Square` type. This is an example of dynamic type safety in C#. To avoid the exception being thrown at runtime, we can include our own type checking code and handle different types in a different way:

```

if (shape is Square)
{
    Square square = ((Square)shape);
    var side = square.Side;
}

if (shape is Circle)
{
    Circle circle = ((Circle)shape);
    var radius = circle.Radius;
}

```

Since `object` is the base type for all reference types in .NET, static type checking can

almost always be circumvented by casting a variable to the object type first, and then to the target type. Using this trick, we can modify the code from our first example to make it compile:

```
string text = "String value";
int textLength = text.Length;
int textMonth = ((DateTime)(object)text).Month;
```

```
DateTime date = DateTime.Now;
int dateLength = ((string)(object)date).Length;
int dateMonth = date.Month;
```

Of course, thanks to dynamic type safety in C#, an `InvalidCastException` will still be thrown. .NET took advantage of this approach to implement common data structures before the introduction of generics while still preserving type safety, albeit only at runtime. These data structures are still available today, although their use is discouraged in favor of their generic alternatives.

```
var list = new ArrayList();
list.Add("String value");
int length = ((string)list[0]).Length;
int month = ((DateTime)list[0]).Month; // no compiler error
```

`ArrayList` is an example of a non-generic data structure in the .NET framework. Since it doesn't provide any type information about the values it contains, we must cast the values back to the correct type on our own before we can use them. If we cast the value to the wrong type as shown above in the line marked with a comment, the compiler can't detect that. The type checking will only happen at runtime.

By using generic data structures instead, the type information is preserved at compile time, allowing full static type checking.

```
var list = new List<string>();
list.Add("String value");
int length = list[0].Length;
int month = list[0].Month; // compiler error
```

There's no need to cast the values anymore, which prevents programmer mistakes and allows the compiler to detect errors.

Looking at the examples so far, C# could be considered a type safe language with full dynamic type safety. However, it contains a language feature which can be used to also circumvent the dynamic type checking. It's called unsafe code.

```
unsafe {
    long longValue = 42;
    long* longPointer = &longValue;
    double* doublePointer = (double*) longPointer;
    double doubleValue = *doublePointer;
}
```

In code blocks marked as unsafe, pointers can be used to gain unrestricted direct access to memory. In the above code sample, we have managed to read a value of type **long** as if it were of type **double**. The resulting value doesn't make any sense in most cases.

Still, both types (long and double) are 64 bits in size. Therefore, we only accessed memory which was previously allocated. Nothing would have prevented us from gaining access to unallocated memory if we haven't taken that precaution ourselves (e.g. if we tried to read a 32-bit int value as a 64-bit double value).

Of course, all of this makes direct memory manipulation and unsafe code in general potentially much more dangerous than regular managed code. That's also the reason why we need to enable it explicitly with a special compiler flag on the *Build* tab of the project *Properties* window.

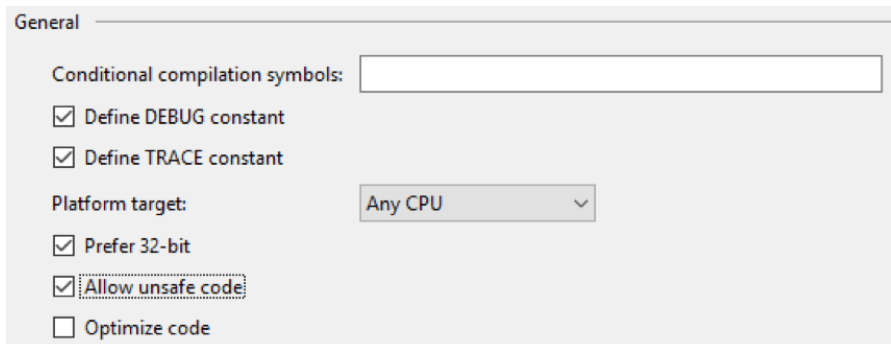


Figure 1: Allow unsafe code in project properties

Except in some rare cases when we need to interoperate with native code or have specific performance requirements, we should avoid using unsafe code.

21

Q21. How can dynamic binding be implemented in C#?

With static binding, the type of each expression is checked at compile time. If the code tries to access a non-existent member of a type, it will result in a compilation error. This behavior is the basis for type safety in a language.

```
var text = "String value";  
var textLength = text.Length; // compiles - Length is a property of  
string  
var lengthString = text.Length.ToString(); // compiles - ToString()  
is a method of int  
var textMonth = text.Month; // doesn't compile - Month is not a  
property of string
```

In contrast, with dynamic binding, the compiler does not do any type checking at compile time. It simply assumes that the code is valid, no matter which member it tries to access on the expression. All the checking is done at runtime, and an exception is thrown if the requested member does not exist on that expression.

Dynamic binding is an important feature of dynamically-typed languages, such as JavaScript. Although C# is primarily considered a statically-typed language with a high level of type safety, it also supports dynamic binding since version 4 when this functionality was added to simplify interaction with dynamically-typed .NET languages such as IronPython and IronRuby introduced at that time.

The core of dynamic binding support in C# is the `dynamic` keyword. With it, variables can be declared to be dynamically-typed. For expressions involving such variables, static type checking at compile time is completely disabled, as in the following example:

```
dynamic text = "String value";
var textLength = text.Length;
var lengthString = text.Length.ToString();
var textMonth = text.Month; // compiles but throws exception at
runtime

dynamic date = DateTime.Now;
var dateLength = date.Length; // compiles but throws exception at
runtime
var dateMonth = date.Month;
```

In the above example, using the `dynamic` keyword doesn't make much sense. It only postpones the checking till runtime, while it could have already been done at compile time if we used static types. It brings no advantages but introduces two significant disadvantages:

- Our code could easily be defective as in the example above because type errors are not detected at compile time.
- Even if there are no defects in code, additional type checks are required at runtime which affects performance.

Of course, there are better use cases for the `dynamic` keyword, otherwise it wouldn't have been added to the language.

For example, it can circumvent the restriction that anonymous types can only be used within a single method. Their type name can't be declared as the return value of a method because it is visible only to the compiler. If the method is declared with a dynamic return value, the compiler will allow it to return a value of an anonymous

type, although the exact type isn't (and can't be) declared explicitly:

```
public dynamic GetAnonymousType()
{
    return new
    {
        Name = "John",
        Surname = "Doe",
        Age = 42
    };
}
```

Of course, the returned value can be used outside the method, with the disadvantage that no IntelliSense or compile time type checking will be available for it as neither the compiler nor the editor is aware of the actual type being returned:

```
dynamic value = GetAnonymousType();
Console.WriteLine($"{value.Name} {value.Surname}, {value.Age}");
```

Since anonymous types are declared as *internal*, this will only work within a single assembly. Code in different assemblies will still be able to get the value when calling the method, but any attempts to access its members will fail with a *RuntimeBinderException*.

However, that's only the beginning of what can be done with the *dynamic* keyword. A great example of how much value dynamic binding can add when used correctly is the JSON.NET library for serializing and deserializing JSON. It can be best explained with the following example:

```
string json = @"
{
    ""name"": ""John"",
    ""surname"": ""Doe"",
    ""age"": 42
}";
```

```
dynamic value = JObject.Parse(json);
Console.WriteLine($"{value.name} {value.surname}, {value.age}");
```


Like the anonymous type example we saw earlier, the **Parse** method returns a value which can be dynamically bound. This allows the JSON object properties to be accessed as the properties of the parsed object although the compiler could have no knowledge about them. We can pass in any JSON object at runtime and still access its properties in code. This functionality couldn't have been implemented with anonymous types.

So, how was it implemented then?

There are two helper objects in the .NET framework which we can use to implement similar functionalities – **ExpandoObject** and **DynamicObject**.

ExpandoObject is the simpler one of the two. It allows members to be added to or removed from any instance of it at runtime. If assigned to a variable declared as dynamic, these members will be dynamically bound at runtime:

```
dynamic person = new ExpandoObject();
person.Name = "John";
person.Surname = "Doe";
person.Age = 42;
```

```
Console.WriteLine($"{person.Name} {person.Surname}, {person.Age}");
```

Members are not limited to being properties. They can be methods as well. To achieve that, a corresponding lambda can be assigned to a member, cast to a matching delegate type. Later, it can be invoked with the standard syntax for calling a method:

```
person.ToString = (Func<string>)(() => $"{person.Name} {person.
Surname}, {person.Age}");
Console.WriteLine(person.ToString());
```

To see which members were added to the **ExpandoObject** at runtime, we can cast the instance to the **IDictionary<string, object>** interface and enumerate its key-value pairs:

```
var dictionary = (IDictionary<string, object>)person;
foreach (var member in dictionary)
{
    Console.WriteLine($"{member.Key} = {member.Value}");
}
```

Using the same interface, we can also remove a member at runtime:

```
dictionary.Remove("ToString");
```

While `ExpandoObject` can be useful in simple scenarios, it gives very little control to the code instantiating and initializing it. The consuming code always has full access to the instance and can modify it to the same extent as the code creating it. Methods are added to it in a similar way to properties, i.e. as lambdas assigned to each individual instance. There is no way to define methods at the class level and automatically make them accessible from every instance.

DynamicObject gives more control to the developer and avoids many of these disadvantages but at the same time requires more code to achieve similar results:

```
class MyDynamicObject : DynamicObject
{
    private readonly Dictionary<string, object> members = new
        Dictionary<string, object>();
    public override bool TryGetMember(GetMemberBinder binder, out
        object result)
    {
        if (members.ContainsKey(binder.Name))
        {
            result = members[binder.Name];
            return true;
        }
        else
        {
            result = null;
            return false;
        }
    }

    public override bool TrySetMember(SetMemberBinder binder, object
        value) {
        members[binder.Name] = value;
        return true;
    }
}
```

With the above implementation, `MyDynamicObject` supports dynamic adding of

properties at runtime just like the `ExpandoObject` does:

```
dynamic person = new MyDynamicObject();
person.Name = "John";
person.Surname = "Doe";
person.Age = 42;

Console.WriteLine($"{person.Name} {person.Surname}, {person.Age}");
```

We can even add methods to it using the same approach: by assigning lambdas as delegates to its members:

```
person.AsString = (Func<string>)(() => $"{person.Name} {person.
Surname}, {person.Age}");
Console.WriteLine(person.AsString());
```

We didn't use the standard method name `ToString` this time because it wouldn't work as expected. If we did, `DynamicObject`'s default `ToString` method would still be called. Unlike `ExpandoObject`, `DynamicObject` first tries to bind all member accesses to its own members and only if that fails, it delegates the resolution to its `TryGetMember` method. Since `DynamicObject` has its `ToString` method just like any other object in the .NET framework, the call will be bound to it and `TryGetMember` won't be called at all.

Thanks to this behavior, we can implement methods directly in our derived class and they will be invoked as expected:

```
public bool RemoveMember(string name)
{
    return members.Remove(name);
}
```

If we need a more dynamic behavior, we can still override the `TryInvokeMember` method. In the following example we will expose all methods of our internal dictionary without having to write specific code for each method. This gives the caller access to it similar to what `ExpandoObject` does.

```
public override bool TryInvokeMember(InvokeMemberBinder binder,
object[] args, out object result)
{
```

```
try
{
    var type = typeof(Dictionary<string, object>);
    result = type.InvokeMember(binder.Name,
        BindingFlags.InvokeMethod | BindingFlags.Public | BindingFlags.
        Instance,
        null, members, args);
    return true;
}
catch
{
    result = null;
    return false;
}
}
```

In the example we just saw, with these methods present in our class, we won't notice any difference between invoking the statically defined `RemoveMember` method and all the dynamically defined methods. If we are using dynamic binding, any methods not defined in our class will be delegated by the `TryInvokeMember` method to the inner dictionary using reflection.

```
person.Remove("AsString");
Console.WriteLine(person.ContainsKey("AsString"));
```

If we wanted to, we could also directly expose the inner dictionary when casting the object to the correct dictionary type just like `ExpandoObject` does. Of course, one way would be to overload the cast operator:

```
public static explicit operator Dictionary<string,
object>(MyDynamicObject instance)
{
    return instance.members;
}
```

If we needed to implement casting in a more dynamic manner, we could always override the `TryConvert` method instead:

```
public override bool TryConvert(ConvertBinder binder, out object
```

```

result)
{
    if (binder.Type.IsAssignableFrom(members.GetType()))
    {
        result = members;
        return true;
    }
    else
    {
        result = null;
        return false;
    }
}

```

Both implementations would give us access to the inner dictionary by casting an instance of the class to the correct type:

```
var dictionary = (Dictionary<string, object>)person;
```

Both the `ExpandoObject` and the `DynamicObject` implement the same `IDynamicMetaObjectProvider` interface to implement the underlying dynamic binding logic. To get full control over the binding process, we can do the same. However, if we can achieve our goal by using either the `ExpandoObject` or `DynamicObject`, they are a better choice because they make our implementation much simpler.

SECTION III

CLASSES AND INHERITANCE

22

Q22. What is polymorphism and how to implement it in C#?

Polymorphism is an important concept in object-oriented languages which allows a piece of code to work with objects of multiple types (classes or structs) without having to know the exact type of each object. If these types implement the same interface, then the calling code can invoke the implementation corresponding to the type of each instance, even if it wasn't known at compile time.

In a role-playing game, there could be a common interface `IWeapon` for all weapons:

```
interface IWeapon
{
    int Damage { get; set; }
    void Attack(IEnemy enemy);
    void Repair();
}
```

Each weapon in the game would then need to implement this interface. A sword could add a `durability` property which would decrease with usage. To restore

durability, the weapon would need to be repaired:

```
class Sword : IWeapon
{
    public int Damage { get; set; }
    public int Durability { get; set; }

    public void Attack(IEnemy enemy)
    {
        if (Durability > 0)
        {
            enemy.Health -= Damage;
            Durability--;
        }
    }

    public void Repair()
    {
        Durability += 100;
    }
}
```

A **Bow** might not need to be repaired at all. However, it will use an arrow for each attack:

```
class Bow : IWeapon
{
    public int Damage { get; set; }
    public int Arrows { get; set; }

    public void Attack(IEnemy enemy)
    {
        if (Arrows > 0)
        {
            enemy.Health -= Damage;
            Arrows--;
        }
    }
}
```



```

    public void Repair()
    { }
}

```

Other code for handling the weapon does not need to know anything about these details. It is only interested in the common interface of all weapons. The **Player** class could hold a reference to the weapon in use and have a method for attacking an enemy with that weapon:

```

class Player
{
    public IWeapon Weapon { get; set; }
    public void Attack(IEnemy enemy)
    {
        Weapon?.Attack(enemy);
    }
}

```

The code would work correctly, no matter which weapon the player had. If the player used a sword, the `Sword.Attack` method would be invoked. If a bow is used, the `Bow.Attack` method would be invoked.

This behavior is not limited only to classes that directly implement the interface. We can imagine a different sword which would mostly behave like our existing sword with only a minor difference.

For example, a poisoned sword would poison the enemy in addition to inflicting direct damage on him. Instead of fully implementing the `IWeapon` interface again, we could derive the `PoisonedSword` class from our existing `Sword` class to avoid repeating the same code in both of them.

For this to work, we first need to mark the **Attack** method in our original `Sword` class as virtual so that the compiler will allow overriding it:

```

public virtual void Attack(IEnemy enemy)
{
    if (Durability > 0)
    {
        enemy.Health -= Damage;
        Durability--;
    }
}

```

```
    }  
}
```

In the derived class, we only need to implement the **Attack** method which is the only one that behaves differently from the base **Sword** class:

```
class PoisonedSword : Sword  
{  
    public override void Attack(IEnemy enemy)  
    {  
        if (Durability > 0)  
        {  
            enemy.Poisoned = true;  
        }  
        base.Attack(enemy);  
    }  
}
```

The code from the base class will still be executed because we are calling `base.Attack`. If we didn't do that, we could have a completely different implementation for the `Attack` method while still keeping everything else in the base class unchanged.

No matter how we try to call the `Attack` method on an instance of `PoisonedSword`, the same code from `PoisonedSword.Attack` will get called:

```
PoisonedSword poisonedSword = new PoisonedSword();  
poisonedSword.Attack(enemy); // will execute PoisonedSword.Attack
```

```
Sword sword = poisonedSword;  
sword.Attack(enemy); // will still execute PoisonedSword.Attack
```

```
IWeapon weapon = poisonedSword;  
weapon.Attack(enemy); // will still execute PoisonedSword.Attack
```

Because of polymorphism, the compile time type of a variable is not important. The code to be executed will always be determined based on its run time type.

23

Q23. What is the meaning of individual accessibility levels?

Each member of a namespace or a type will always be visible to other members *within* the same scope. Accessibility levels are used to specify who else can see (and access) these members *outside* their own scope.

Namespaces can only contain types (classes, structs, interfaces, enums and delegates). Only two different accessibility levels are supported for members inside namespaces:

Internal members are only visible within their own assembly, i.e. their own compilation unit: library (.dll) or executable (.exe).

Public members don't have this restriction: they are also visible from other assemblies which must of course reference their assembly to access them.

```
namespace SampleTypes
{
    public class ExternallyVisibleType
    {
```

```
    // visible in other assemblies
}

internal class HiddenType
{
    // visible only in its own assembly
}
}
```

When no access modifier is specified for a member, its accessibility level will default to **internal**. However, it's a good idea to always clearly specify the intended behavior by explicitly using the desired access modifier.

Members placed inside classes support several additional accessibility levels in comparison to those placed directly inside namespaces.

Most class members (e.g. methods, properties, fields, events, etc.) cannot be placed outside classes, structs or interfaces. However, other types can be nested inside classes. The following basic accessibility levels are supported for all members:

Private members are only visible to other members within the same class.

Protected members expand their visibility to members in classes which derive from their own class.

Internal members are only visible inside their own assembly. This is true even if their parent class is declared as **public**: the class will then be visible in other assemblies, but the internal member won't.

Public members have no visibility restrictions. However, they can only be accessed where their parent class is visible as well.

```
public class SampleType
{
    private void ClassSpecificMethod()
    {
        // visible only inside the SampleType class
    }

    protected void MethodForDerivedClasses()
```

```

{
    // visible inside the SampleType class and any class deriving
    from it
}

internal void AssemblySpecificMethod()
{
    // visible in its own assembly only
}

public void FullyVisibleMethod()
{
    // visible everywhere
}
}

```

There are two additional accessibility levels. Each one of them is specified using two access modifiers:

- **Protected internal** members are visible inside every class deriving from their class just like protected members. But they are also visible throughout their own assembly, even in classes which don't derive from their class.
- **Private protected** members are much more restricted. They are only visible inside classes which derive from their own class. But not all of them. These classes must also be defined in the same assembly. Private protected members won't be accessible inside classes in other assemblies, which are also deriving from this class. This accessibility level was introduced in C# 7.2 and is not supported in earlier versions of the language.

In a way, 'protected internal' and 'private protected' accessibility levels are similar. They both modify the behavior of the protected accessibility level:

- Protected internal expands the visibility to all types within the same assembly in addition to all derived types.
- Private protected restricts the visibility to only those derived types which are defined within the same assembly.

When no access modifier is set for a member in a class, its visibility will be set to

private by default. As for members inside namespaces, it's a good practice to always specify the desired accessibility level explicitly and make the intention clear to anyone reading the code.

Other types are not as flexible as classes. As a result, they don't support some accessibility levels because these wouldn't make sense:

- **Structs** cannot be derived from. Therefore, the accessibility levels which define different visibility for derived types are not supported. Hence, the only access modifiers supported are private, internal and public.
- **Interfaces** and **enums** can only contain public members. Hence, they don't allow using any access modifier, not even public, as all members are public by default anyway.

```
public interface SampleInterface
{
    // all members are public
    void PublicMember();
    void AnotherPublicMember();
}

public enum SampleEnum
{
    // all members are public
    PublicValue,
    AnotherPublicValue
}
```

- **Delegates** don't contain any other members to apply access modifiers to.

Setting the correct accessibility level for each member of a type or namespace is important because this allows hiding internal implementation details from other code using the member's parent (either type or namespace).

If some code can't access a member, it won't be able to use it directly and take a dependency on it. The less accessible a member is, the less code will be dependent on it and the easier it will be to change, no matter whether the change affects just the behavior, or also the member signature. The member could even be deleted altogether.

When no code outside the current assembly can access a member (this is true for private, internal and private protected members), then all the code that's using it can be changed at the same time as the member itself. This makes the changes possible without having to think about backward compatibility at the level of that method. Taking this into consideration, it's best to always use the strictest accessibility level possible for every member.

There are two more language features affecting member accessibility:

- Using the assembly level attribute **InternalsVisibleToAttribute**, an assembly can specify that all its internal (and protected internal) members will be accessible from another assembly which is specified with this attribute at compile time (by name, or by name and public key token for signed assemblies). Using this attribute is usually considered a bad practice. In certain scenarios, it might make sense to give such access to an assembly containing tests, although it should also be possible (and often better) to test internal members through other methods calling them, which are accessible from outside their assembly.

```
// can be placed in any file compiled into the assembly
[assembly: InternalsVisibleTo("TestAssembly")]
```

- Using reflection, members can be accessed and invoked without regard to their accessibility level. However, it is very dangerous to directly access members which are considered hidden, as they can change any time without warning (especially when the code is not under our control). On top of the negative performance implications, this is another reason to avoid reflection whenever possible.

24

Q24. What are the advantages and disadvantages of abstract classes over interfaces?

Interfaces are the basis for polymorphism in C#. They specify a collection of members (methods, properties, events and indexers) with signatures only, i.e. without bodies.

```
interface ISampleInterface
{
    int Method(int arg);
    int Property { get; set; }
    event EventHandler<EventArgs> MethodInvoked;
    string this[int index] { get; set; }
}
```

These members must be implemented by every class implementing that interface.

```
class SampleClass : ISampleInterface
{
    public int Method(int arg)
    {
```



```

        OnMethodInvoked();
        return arg * 2;
    }

    public int Property { get; set; }

    public event EventHandler<EventArgs> MethodInvoked;

    private void OnMethodInvoked()
    {
        MethodInvoked?.Invoke(this, new EventArgs());
    }

    private Dictionary<int, string> indexerDictionary = new
    Dictionary<int, string>();
    public string this[int index]
    {
        get
        {
            return indexerDictionary[index];
        }
        set
        {
            indexerDictionary[index] = value;
        }
    }
}

```

However, interfaces don't contain any code that could be reused by different classes implementing the interface. Each implementing class must contain all the required code itself.

To avoid repeating the same code in similar classes implementing the interface, abstract classes can be used.

```

abstract class SampleAbstractClass : ISampleInterface
{
    public abstract int MethodImplementation(int arg);
    public int Method(int arg)

```

```
{
    OnMethodInvoked();
    return MethodImplementation(arg);
}

public virtual int Property { get; set; }

public event EventHandler<EventArgs> MethodInvoked;
protected virtual void OnMethodInvoked()
{
    MethodInvoked?.Invoke(this, new EventArgs());
}

protected Dictionary<int, string> indexerDictionary = new
Dictionary<int, string>();
public virtual string this[int index]
{
    get
    {
        return indexerDictionary[index];
    }
    set
    {
        indexerDictionary[index] = value;
    }
}
}
```

Of course, any member marked as **virtual** can be overridden in a class deriving from this abstract class, thereby allowing the existing code to be reused, enhanced or replaced by different code.

The main distinguishing feature of abstract classes is, just like interfaces, they cannot be instantiated directly. This allows them to contain abstract methods which just like methods in interfaces, only specify the signature and don't have an implementation. Any deriving non-abstract class is required to provide an implementation for any abstract methods in its base abstract class. These derived non-abstract classes are the ones that are then instantiated.

Although abstract classes don't need to implement an interface, they most often do. Deriving from an abstract class instead of an interface directly is usually simpler and requires less code to be written because a lot of it is already included in the base abstract class.

```
class SampleImplementingClass : SampleAbstractClass
{
    public override int MethodImplementation(int arg)
    {
        return arg * 2;
    }
}
```

At a quick glance, abstract classes seem to be a more flexible alternative to interfaces: they can still declare methods without implementations, but can also include virtual members with existing code which can be reused or replaced, as well as non-virtual members.

However, there's a very important limitation to abstract classes. Since C# doesn't support multiple inheritance, a class can only derive from a single base abstract class. There's no such limitation for interfaces – classes can implement any number of different interfaces.

25

Q25. When should one explicitly implement an interface member?

Usually interface members are implemented in a class as if they were regular class members. This is called *implicitly* implementing an interface member.

```
public interface ISimpleInterface
{
    void Method();
}

public class ImplicitImplementation : ISimpleInterface
{
    public void Method()
    {
        Console.WriteLine("Implicit implementation");
    }
}
```

Such a member is accessible on a class directly, just like any other member of the

class.

```
var instance = new ImplicitImplementation();
instance.Method(); // outputs: Implicit implementation
```

It can still be accessed after casting the class to the interface.

```
var instance = new ImplicitImplementation();
var asInterface = (ISimpleInterface)instance;
asInterface.Method(); // outputs: Implicit implementation
```

That's not the only way to implement an interface member. Alternatively, it can be implemented *explicitly*. The syntax in this case is a bit different.

```
public class ExplicitImplementation : ISimpleInterface
{
    void ISimpleInterface.Method()
    {
        Console.WriteLine("Explicit implementation");
    }
}
```

The behavior of explicitly implemented interface members is also different. They aren't treated as regular class members and therefore aren't accessible directly.

```
var instance = new ExplicitImplementation();
instance.Method(); // will not build
```

The only way to access an explicitly implemented interface member is to cast an instance of the class it is implemented in, to the interface it belongs to.

```
var instance = new ExplicitImplementation();
var asInterface = (ISimpleInterface)instance;
asInterface.Method(); // outputs: Explicit implementation
```

This allows members to be hidden when they aren't relevant to the implementing class and only make sense to be called as part of the interface. It also allows a class to have two different methods with the same name: one accessible as a class member, and the other one accessible as an interface member.

```
public class ClassMemberAndExplicitImplementation :
ISimpleInterface
{
    public void Method()
    {
        Console.WriteLine("Regular class member");
    }

    void ISimpleInterface.Method()
    {
        Console.WriteLine("Explicit implementation");
    }
}

var instance = new ClassMemberAndExplicitImplementation();
instance.Method(); // outputs: Regular class member

var asInterface = (ISimpleInterface)instance;
asInterface.Method(); // outputs: Explicit implementation
```

The possibilities of explicitly implemented members doesn't stop here. A class can implement multiple interfaces containing a member with the same signature. If the members were implemented implicitly, then a single method could be used for implementing all of them.

```
public interface ISecondInterface
{
    void Method();
}

public class ImplicitTwoInterfaces : ISimpleInterface,
ISecondInterface
{
    public void Method()
    {
        Console.WriteLine("Implicit implementation");
    }
}
```

```
var instance = new ImplicitTwoInterfaces();
var asSimpleInterface = (ISimpleInterface)instance;
asSimpleInterface.Method(); // outputs: Implicit implementation
```

```
var asSecondInterface = (ISecondInterface)instance;
asSecondInterface.Method(); // outputs: Implicit implementation
```

By explicitly implementing the interface members, a different implementation can be provided for each interface.

```
public class ExplicitTwoInterfaces : ISimpleInterface,
ISecondInterface
{
    void ISimpleInterface.Method()
    {
        Console.WriteLine("ISimpleInterface");
    }

    void ISecondInterface.Method()
    {
        Console.WriteLine("ISecondInterface");
    }
}
```

```
var instance = new ExplicitTwoInterfaces();
var asSimpleInterface = (ISimpleInterface)instance;
asSimpleInterface.Method(); // outputs: ISimpleInterface
```

```
var asSecondInterface = (ISecondInterface)instance;
asSecondInterface.Method(); // outputs: ISecondInterface
```

Explicit implementation of interface members isn't used often, but it does provide greater flexibility when needed, which would otherwise be much more difficult to achieve.

26

Q26. What does the "new" modifier on a method mean?

When accessing members on derived classes, the code executed usually depends only on the runtime type of the object instance. It doesn't matter whether the instance is stored in a variable of its runtime type, or in a variable of its base type. The same code is executed in both cases – the one from its runtime type.

```
public class BaseClass
{
    public virtual void Method()
    {
        Console.WriteLine("From BaseClass");
    }
}

public class DerivedClass : BaseClass
{
    public override void Method()
    {
```



```

        Console.WriteLine("From DerivedClass");
    }
}

var instance = new DerivedClass();
instance.Method(); // output: From DerivedClass

var asBase = (BaseClass)instance;
asBase.Method(); // output: From DerivedClass

```

There is a way to change this default behavior.

When the `new` modifier is used on a member in the derived class, it hides the implementation inherited from the base class, and with that, effectively breaks the inheritance chain:

- The member inherited from the base class will be hidden in the derived class, therefore the derived class implementation will be invoked on a variable (or expression) with the type of the derived class.
- The member in the derived class will not override the one from the base class, therefore the base class implementation will be invoked when the same instance of the derived class is cast to the base type.

The following piece of code manifests this behavior:

```

public class NewModifierClass : BaseClass
{
    public new void Method()
    {
        Console.WriteLine("From NewModifierClass");
    }
}

var instance = new NewModifierClass();
instance.Method(); // output: From NewModifierClass

var asBase = (BaseClass)instance;
asBase.Method(); // output: From BaseClass

```

Because of such unexpected behavior, it is generally not recommended to use the **new** modifier. It will cause a method call to result in invoking a different method based on whether the instance it is invoked on is cast to the base type, or to the derived type. To most, this is unintuitive and difficult to explain without carefully examining the source code.

Instead of using the **new** modifier, it's better to simply override the method from the base class. However, if that method is not **virtual**, it cannot be overridden. In that case, using the **new** modifier is the only way for the derived class to define a different piece of code to get executed instead of that from the base class.

```
public class NonVirtualBaseClass
{
    public void Method()
    {
        Console.WriteLine("From NonVirtualBaseClass");
    }
}
```

Since **Method** in **NonVirtualBaseClass** is not **virtual**, it cannot be overridden. Any attempt to do so will not compile:

```
public class NonVirtualDerivedClass : NonVirtualBaseClass
{
    public override void Method() // will not compile
    {
        Console.WriteLine("From DerivedClass");
    }
}
```

It's still possible to use the **new** modifier on the method with the same name in the derived class:

```
public class NonVirtualDerivedClass : NonVirtualBaseClass
{
    public new void Method() // will still compile
    {
        Console.WriteLine("From NonVirtualDerivedClass");
    }
}
```

However, this will result in a different behavior than overriding the method would:

```
var instance = new NonVirtualDerivedClass();  
instance.Method(); // output: From NonVirtualDerivedClass  
  
var asBase = (NonVirtualBaseClass)instance;  
asBase.Method(); // output: From NonVirtualBaseClass
```

The resulting behavior will be atypical and error-prone, but since there is no better alternative available, it might sometimes be necessary to use this approach.

27

Q27. How to correctly override the Equals method?

The `Equals` method is one of the few methods that's available for every type in the .NET framework. This is because it's defined on the `Object` type which is the implicit base type for all types. As such, it is being used by many types in the base class library, especially collections. Therefore, additional caution is required if you decide to override the method yourself.

There are two different default implementations provided for the `Equals` method:

For value types, the method checks for value equality of all their fields, i.e. two variables are equal if all their fields have the same value:

```
public struct PointStruct
{
    public double X { get; set; }
    public double Y { get; set; }
}
var point1 = new PointStruct { X = 1, Y = 2 };
```

```
var point2 = new PointStruct { X = 1, Y = 2 };
Assert.IsTrue(point1.Equals(point2));
```

For reference types, the method checks for reference equality, i.e. two variables are equal if they point at the same instance in memory:

```
public class PointClass
{
    public double X { get; set; }
    public double Y { get; set; }
}
```

```
var point1 = new PointClass { X = 1, Y = 2 };
var point2 = point1;
Assert.IsTrue(point1.Equals(point2));
```

Two different instances of reference types are not equal even if their fields have the same value:

```
var point1 = new PointClass { X = 1, Y = 2 };
var point2 = new PointClass { X = 1, Y = 2 };
Assert.IsFalse(point1.Equals(point2));
```

The most typical reason for overriding the Equals method is to implement value equality for reference types based on all or some of the fields in the type:

```
public override bool Equals(object obj)
{
    PointClass point = obj as PointClass;
    if (point == null)
    {
        return false;
    }
    else
    {
        return X.Equals(point.X) && Y.Equals(point.Y);
    }
}
```

```
var point1 = new PointClass { X = 1, Y = 2 };
```

```
var point2 = new PointClass { X = 1, Y = 2 };  
Assert.IsTrue(point1.Equals(point2));
```

While checking for value equality by overriding the Equals method, you cannot check whether two variables point at the same instance. If you still need to check that, you can use the static Object.ReferenceEquals method:

```
var point1 = new PointClass { X = 1, Y = 2 };  
var point2 = new PointClass { X = 1, Y = 2 };  
Assert.IsFalse(ReferenceEquals(point1, point2));
```

```
var point3 = point1;  
Assert.IsTrue(ReferenceEquals(point1, point3));
```

When overriding the Equals method, there are many requirements that need to be met because existing code relies on them.

Most of these requirements are based on math, as they originate from the formal definition of the **equivalence relation**, which is required to be:

- **Reflexive**, i.e. any variable must be equal to itself, meaning that `a.Equals(a)` must always return `true`.
- **Symmetric**, i.e. the result does not change when the variables being compared are swapped. The result of `a.Equals(b)` must always be the same as the result of `b.Equals(a)`.
- **Transitive**, i.e. when a variable is equal to two other variables, then those two variables are also equal to each other. This means that if both `a.Equals(b)` and `b.Equals(c)` return `true`, then `a.Equals(c)` must also return `true`.

In addition to that, the Equals method must be **deterministic**, i.e. its result depends solely on its input values. Hence, the call to `a.Equals(b)` must always return the same result if neither `a` nor `b` were modified.

For the sake of completeness, there are two more edge cases specified:

- No non-null value may be equal to a null value, i.e. `a.Equals(null)` must always return `false`.
- Two NaN (not a number) values are always equal to each other, i.e. `a.Equals(b)`

must return `true` if both a and b represent a NaN value.

The final requirement when overriding the Equals method is less intuitive: whenever you override the Equals method, you must also override the GetHashCode method in such a way that it will return the same result for all variables which are treated as equal by the Equals method implementation. Failing to do so will result in erratic behavior of types in the base class library which use the GetHashCode method as a faster way to test whether two values might be equal:

```
var point1 = new PointClass { X = 1, Y = 2 };
var point2 = new PointClass { X = 1, Y = 2 };

// overridden Equals method returns true
Assert.IsTrue(point1.Equals(point2));
// default GetHashCode method returns different values
Assert.IsFalse(point1.GetHashCode() == point2.GetHashCode());

var set = new HashSet<PointClass>();
set.Add(point1);
set.Add(point2);
// HashSet adds both points because of different GetHashCode value
Assert.IsFalse(set.Count == 1);
```

The HashSet collection implements the behavior of a set. Therefore, when adding the same value to the set twice, it should only be added once. Since point1 and point2 are equal according to the Equals method, they should only be added once. However, because the default implementation of the GetHashCode method returns a different value for each one of them (based on their reference), both values are added. This happens because before calling the presumably slower Equals method, the GetHashCode method is called for both values. Since the results are different, the values are assumed not to be equal. Therefore, in the example we just saw, the Equals method wasn't called at all.

The simplest way to implement a compliant GetHashCode method for value equality is by using the bitwise exclusive OR (^) operation to combine the values returned by individual GetHashCode calls for fields included in the value equality:

```
public override int GetHashCode() {
    return X.GetHashCode() ^ Y.GetHashCode();
}
```

This implementation fixes the beforementioned problem. It returns the same value for points which are equal according to value equality, however it doesn't do a very good job at returning different results for points, which aren't equal. For example, the result will be the same for all points with the same value for their X and Y coordinates (because $n \wedge n = 0$ for any n of type `int`) which isn't an uncommon scenario.

Therefore, a different algorithm is usually recommended:

```
public override int GetHashCode()
{
    unchecked
    {
        return 17 + X.GetHashCode() * 23;
    }
}
```

This algorithm is also used by the C# compiler for anonymous types. It does a much better job at returning different values for different points and therefore makes the `GetHashCode` result a much better predictor of whether the two points are going to be equal or not. It's important that the two seemingly magic numbers are prime numbers. The algorithm will work just as well for other prime numbers. The `unchecked` block around the expression disables overflow checking, so that no `OverflowException` will be thrown if the result of the arithmetic operation is outside of the range of values supported by the `int` data type.

To avoid implementing your own algorithm, you can also take advantage of the one for the `ValueTuple` type. Since C# 7 where they were introduced, a temporary tuple can be created and its `GetHashCode` value can be returned:

```
public override int GetHashCode()
{
    return (X, Y).GetHashCode();
}
```

Since `ValueTuple` is a struct, there's no memory allocation involved although a new instance is created.

All suggested implementations for the `GetHashCode` method will fix the problem with the `HashSet` type:


```

var point1 = new PointClass { X = 1, Y = 2 };
var point2 = new PointClass { X = 1, Y = 2 };

Assert.IsTrue(point1.Equals(point2));
Assert.IsTrue(point1.GetHashCode() == point2.GetHashCode());

var set = new HashSet<PointClass>();
set.Add(point1);
set.Add(point2);
Assert.IsTrue(set.Count == 1);

```

The result of GetHashCode will now be the same for both points. Therefore, the point will only be added to the set once, as expected.

Because of how the GetHashCode method is used by collection types, it's dangerous to override the GetHashCode method and the Equals method with value equality for reference types which are mutable, i.e. which can change and thus return different GetHashCode results after they have already been added to the collection.

A type can implement the IEquatable<T> interface to add a strongly typed Equals method. When doing so, it is strongly suggested that the generic strongly-typed IEquatable<T>.Equals method always returns the same result as the overridden Object.Equals method does:

```

public bool Equals(PointClass other)
{
    if (other == null)
    {
        return false;
    }
    else
    {
        return X.Equals(other.X) && Y.Equals(other.Y);
    }
}

public override bool Equals(object obj) {
    PointClass point = obj as PointClass;
    return Equals(point);
}

```

Types with different implementations for the two Equals methods can manifest surprising behavior which can quickly result in subtle bugs. An example of such a class is the StringBuilder class:

```
var builder1 = new StringBuilder("string");
var builder2 = new StringBuilder("string");
Assert.IsTrue(builder1.Equals(builder2));

var asObject1 = (object)builder1;
Assert.IsFalse(asObject1.Equals(builder2));
```

StringBuilder implements the strongly-typed `IEquatable<StringBuilder>.Equals` method with value equality semantics. However, it doesn't override the `Object.Equals` method and keeps its reference equality semantics. As a result, the former returns `true` for two different StringBuilder instances with the same string contents, while the latter returns `false`.

28

Q28. When and how to overload operators?

Operator overloading in C# is mostly just syntactic sugar. Instead of calling instance or static methods on a type, an overloaded operator can be used:

```
var complex1 = new ComplexNumber(2, 1);  
var complex2 = new ComplexNumber(1, 2);  
  
var result1 = ComplexNumber.Add(complex1, complex2); // 3+3i  
var result2 = complex1 + complex2; // 3+3i
```

Of course, overloading operators only makes sense in very specific scenarios. For anyone who is familiar with the concept of [complex numbers](#), adding two complex numbers using an overloaded operator would be just as easy if not easier to understand, than using the static method.

On the other hand, overloading the same operator for a different type (e.g. a Person type) would be more confusing than helpful. The addition operation has no inherent meaning for a person and any developer reading the code would be forced to check

the implementation of the overloaded operator to understand what is happening.

This is the most important criteria to keep in mind when deciding whether to overload an operator for a specific type or not. **You should only do it when this will contribute to the readability and comprehension of the code.** If there is no existing or agreed upon meaning for an operator in the context of a type, it's a better idea to stick with ordinary methods which can better express their meaning through their name.

The `ComplexNumber` type from the above example could be implemented as follows:

```
public struct ComplexNumber
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    public ComplexNumber(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }
}
```

The static method for adding two complex numbers adds the values for each component separately:

- the real components of both numbers and
- the imaginary components of both numbers.

```
public static ComplexNumber Add(ComplexNumber complex1,
ComplexNumber complex2)
{
    var real = complex1.Real + complex2.Real;
    var imaginary = complex1.Imaginary + complex2.Imaginary;
    return new ComplexNumber(real, imaginary);
}
```

To overload the `+` operator instead, only the signature of the method needs to change:

```
public static ComplexNumber operator +(ComplexNumber complex1,
ComplexNumber complex2)
{
    var real = complex1.Real + complex2.Real;
    var imaginary = complex1.Imaginary + complex2.Imaginary;
    return new ComplexNumber(real, imaginary);
}
```

Operator overloads are in fact static methods inside the type for which they are defined. The only difference from a regular static method is that instead of the method name (e.g. Add), the operator keyword is used, followed by the operator to overload (e.g. operator + in the above example).

Only the first parameter of the operator overload method must match the type for which it is defined. The second parameter can be of a different type. For a complex number, adding a real number of type `double` could be supported as well:

```
public static ComplexNumber operator +(ComplexNumber complex,
double real)
{
    return new ComplexNumber(complex.Real + real, complex.Imaginary);
}
```

```
var complex1 = new ComplexNumber(2, 1);
var complex = complex1 + 5; // 7+i
```

Unary operators can be overloaded as well. In such a case, the overload method only has a single parameter:

```
public static ComplexNumber operator -(ComplexNumber complex)
{
    return new ComplexNumber(-complex.Real, -complex.Imaginary);
}
```

```
var complex1 = new ComplexNumber(2, 1);
var complex = -complex1; // -2-i
```

C# does not support overloading of all operators. Only a subset of them can be overloaded directly using a static overload method as in the above examples:

- Arithmetic operators: + (binary and unary), ++, - (binary and unary), --, *, /, and %.
- Bitwise operators: &, |, ^, ~, <<, and >>.
- Logical operators: !, true, and false. The latter two return true when an instance of the type represents a true or false value, respectively.
- Comparison operators: ==, !=, <, >, <=, and >=.

The comparison operators must be overloaded in pairs, e.g. when == is overloaded, != must be overloaded as well. The same goes for < and >, or <= and >=. In many cases, the best way to implement the second operator overload in the pair is by calling the first operator overload:

```
public static bool operator ==(ComplexNumber complex1,
ComplexNumber complex2)
{
    return complex1.Real == complex2.Real && complex1.Imaginary ==
complex2.Imaginary;
}
```

```
public static bool operator !=(ComplexNumber complex1,
ComplexNumber complex2)
{
    return !(complex1 == complex2);
}
```

```
var complex1 = new ComplexNumber(2, 1);
var complex2 = new ComplexNumber(2, 1);
var equal = complex1 == complex2; // true

var complex3 = new ComplexNumber(1, 2);
var notEqual = complex1 != complex3; // true
```

When overloading the equality operator, it's highly recommended to also override the Equals and GetHashCode methods for the same type.

With the overloadable operators listed so far, several additional operators will be indirectly overloaded because they call one of the above operators in their implementation:

- Assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, and >>=) call the corresponding binary arithmetic or bitwise operator. For example, the += operator will call the overloaded + operator. Therefore, with the overload created above, the following piece of code will also automatically work as expected:

```
var complex = new ComplexNumber(2, 1);
var complex1 = new ComplexNumber(1, 2);
complex += complex1; // 3+3i
```

- Short circuiting logical operators (&&, and ||) use the matching eagerly evaluated logical operators (&, and |, respectively) to combine the two values, and the true and false operators to evaluate whether the first operand is true or false. The operation is always evaluated lazily, i.e. when the result can already be determined from the first operand, the overload method for the binary operator will not be called. It rarely makes sense to overload these operators. One use case is the implementation of tri-state Boolean values (true, false, null), e.g. to represent an equivalent database type.

A special case is the *cast* operator. Its implementation calls the matching conversion operator for the type. To cast an instance of the ComplexNumber type to a double, we need to implement the following explicit conversion operator:

```
public static explicit operator double(ComplexNumber complex)
{
    return complex.Real;
}
```

It will return only the real component of a complex number. This allows us to perform the following cast:

```
var complex = new ComplexNumber(2, 1);
var real = (double)complex; // 2
```

If we implemented the implicit conversion operator instead (using the **implicit** keyword in place of the **explicit** keyword), we wouldn't even need to use the cast operator. It would be enough to assign a ComplexNumber instance to a variable of type double:

```
double real = complex; // 2
```

However, this would not be a good idea in my opinion. Using the cast operator explicitly expresses the intent of the code much better, particularly when the operation is not reversible as in our case.

29

Q29. How do delegates and events differ?

Delegates are special types which describe a method signature, consisting of its parameters and the return type.

```
public delegate int Transform(int n);
```

When a variable of that delegate type is declared in code, any static or instance method matching its signature can be assigned to it.

```
int Increment(int n)
{
    return n + 1;
}
```

```
Transform incrementFunction = Increment;
var incremented = incrementFunction(1); // 2
```

In effect, a variable of a delegate type is a strongly-typed reference to a method which

can be invoked or passed around as a parameter to another method:

```
void TransformAll(List<int> list, Transform transform)
{
    for (var i = 0; i < list.Count; i++)
    {
        list[i] = transform(list[i]);
    }
}

var list = new List<int> { 1, 2, 3, 4, 5 };
TransformAll(list, Increment); // list = { 2, 3, 4, 5, 6 }
```

The TransformAll method can be easily reused to apply different transformations to a list of numbers by passing it a different transform method:

```
int Decrement(int n)
{
    return n - 1;
}

var list = new List<int> { 1, 2, 3, 4, 5 };
TransformAll(list, Decrement); // list = { 0, 1, 2, 3, 4 }
```

Creating a regular named method on a type for the sole reason to pass it as an argument to another method, or to assign to a variable without ever calling it otherwise, might seem unnecessary. That's why later versions of C# introduced an alternative syntax for writing code blocks which will only ever be assigned to a delegate type and never invoked directly.

In C# 2, *anonymous methods* were introduced to avoid the need for creating a named method as a type member. They allow a method to be directly defined inline, where it is assigned to a variable or passed as an argument:

```
Transform incrementFunction = delegate(int n)
{
    return n + 1;
};
```

```

var incremented = incrementFunction(1); // 2
var list = new List<int> { 1, 2, 3, 4, 5 };
TransformAll(list, delegate(int n)
{
    return n - 1;
}); // list = { 0, 1, 2, 3, 4 }

```

In C# 3, lambda expressions were introduced as an even shorter alternative syntax for defining an inline block of code:

```

Transform incrementFunction = n =>
{
    return n + 1;
};
var incremented = incrementFunction(1); // 2

```

Using this syntax, the lambda expression could consist of multiple statements just like an anonymous or named method. Its only requirement is that it must return a value matching the return type declared by the delegate type.

When the inline code consists of only a single statement or an expression, an even shorter lambda expression syntax can be used:

```

Transform incrementFunction = n => n + 1;
var incremented = incrementFunction(1); // 2

```

To reduce the overhead of using delegates even further, a selection of generic delegate types was added to the base class library in .NET framework 3.5, mostly removing the need to create your own delegate types. Instead of the Transform delegate type, we can use Func<int, int>:

```

void ProcessAll(List<int> list, Func<int, int> processFn)
{
    for (var i = 0; i < list.Count; i++)
    {
        list[i] = processFn(list[i]);
    }
}

```

```
var list = new List<int> { 1, 2, 3, 4, 5 };  
ProcessAll(list, x => x - 1); // list = { 0, 1, 2, 3, 4 }
```

Both anonymous methods and lambda expressions can be used to instantiate delegates. However, it is recommended to use lambda expressions instead of anonymous or named methods unless you are targeting an older version of C# and the .NET framework.

Anonymous methods and lambda expressions have another potential advantage over named methods. The code inside them has access to all variables outside it which are in scope i.e. where the anonymous method or lambda expression is declared:

```
var list = new List<int> { 1, 2, 3, 4, 5 };  
  
var increment = 1;  
ProcessAll(list, x => x + increment); // list = { 2, 3, 4, 5, 6 }  
  
increment = 3;  
ProcessAll(list, x => x + increment); // list = { 5, 6, 7, 8, 9 }
```

Delegates can be combined. Multiple methods can be invoked with a single invocation of delegate-typed variable. To add a method to a delegate variable or remove one from it, the `+` and `-` operators are used respectively:

```
Action firstDelegate = () =>  
{  
    Console.WriteLine("First delegate invoked.");  
};  
Action secondDelegate = () =>  
{  
    Console.WriteLine("Second delegate invoked.");  
};  
  
Action combinedDelegate = firstDelegate;  
Console.WriteLine("First call.");  
combinedDelegate();  
  
combinedDelegate += secondDelegate;  
Console.WriteLine("Second call.");  
combinedDelegate();
```

```
combinedDelegate -= firstDelegate;
Console.WriteLine("Third call.");
combinedDelegate();
```

Between invocations of `combinedDelegate` in the above code, we are adding and removing lambdas. The resulting output will be as follows:

```
First call.
First delegate invoked.
Second call.
First delegate invoked.
Second delegate invoked.
Third call.
Second delegate invoked.
```

This capability of invoking multiple methods with a single delegate is used in events which provide an abstraction layer on top of delegates to give them publish and subscribe semantics. Events are exposed as public members of types. Other instances can subscribe to or unsubscribe from them.

To subscribe to an event, the same syntax is used as that for combining delegates. To unsubscribe from it, the syntax is the same as for removing a component delegate from a combined delegate:

```
var instance = new NotifyingClass();
EventHandler<MethodEventArgs> handler = (sender, eventArgs) =>
{
    Console.WriteLine($"Event handler called with {eventArgs.
Argument}.");
};
instance.MethodInvoked += handler; // subscribe to event
instance.MethodInvoked -= handler; // unsubscribe from event
```

Although not required, it is strongly recommended that all events use the `EventHandler` delegate type when no arguments are required, or its generic version `EventHandler<T>` when the delegate needs to be invoked with additional arguments. The generic type for the arguments should be derived from the `EventArgs` base class. Therefore, when implementing a custom event, a custom `EventArgs`-derived class will need to be implemented first:

```
public class MethodEventArgs : EventArgs
{
    public int Argument { get; private set; }
    public MethodEventArgs(int argument)
    {
        Argument = argument;
    }
}
```

The event can then be declared using the `EventHandler<MethodEventArgs>` delegate type:

```
public class NotifyingClass
{
    public event EventHandler<MethodEventArgs> MethodInvoked;
    public void Method(int argument)
    {
        MethodInvoked?.Invoke(this, new MethodEventArgs(argument));
    }
}
```

The `Method` method invokes the event handler using the null-conditional operator which was introduced in C# 6. If no event handler was subscribed to the event, the value of `MethodInvoked` would be null and invoking it without the null-conditional operator would therefore result in a `NullReferenceException`. The operator accesses the event only once, so that the value can't be changed from another thread between the null check and the invocation.

Apart from standardizing the publish and subscribe interaction, events also provide an abstraction over directly exposing a delegate. They make it possible to manually implement the logic for handling the addition and removal of event handlers:

```
public class InstrumentedNotifyingClass
{
    private EventHandler<MethodEventArgs> methodInvokedDelegate;
    public event EventHandler<MethodEventArgs> MethodInvoked
    {
        add
        {
```

```

        methodInvokerDelegate += value;
        Console.WriteLine("Added event handler.");
    }

    remove
    {
        methodInvokerDelegate -= value;
        Console.WriteLine("Removed event handler.");
    }
}

public void Method(int argument)
{
    methodInvokerDelegate?.Invoke(this, new
        MethodEventArgs(argument));
}
}

```

The code above adds logging to both operations. We can test it with the following code:

```

var instance = new InstrumentedNotifyingClass();
EventHandler<MethodEventArgs> handler = (sender, eventArgs) =>
{
    Console.WriteLine($"Event handler called with {eventArgs.
Argument}.");
};

instance.MethodInvoker += handler;
instance.Method(42);
instance.MethodInvoker -= handler;

```

The output to console will reflect the exact order of calls:

```

Added event handler.
Event handler called with 42.
Removed event handler.

```

Although the events can be customized in such a way, it is rarely done in production code and you should avoid it unless you have a good reason for it.

30

Q30. What's special about the IDisposable interface?

In the .NET framework, the garbage collector is the one responsible for freeing the memory allocated by applications.

Although the allocating and releasing of memory is managed by the .NET framework, applications might still require other resources (e.g. files, database connections, etc.). These are not managed and are therefore not automatically freed by the garbage collector once they are not used any more. The application code is responsible for freeing any such unmanaged resources.

When unmanaged resources are accessed via types in the base class framework (e.g. `StreamReader` for reading text files), these types also incorporate the code for releasing any resources they have allocated. If you don't handle this explicitly in your code, these unmanaged resources will be released when the garbage collector collects the corresponding type instance and releases its memory.

Since the garbage collector doesn't have any knowledge of the unmanaged resources handled by that type, it will trigger the garbage collection solely based on the

managed memory consumption. This might not happen immediately if there's still enough memory available. However, the unmanaged resources might be scarcer and should be released as soon as they aren't needed any more, without waiting for the garbage collector to run (e.g. files shouldn't be locked for writing any longer than they are needed).

To enable that, types which access unmanaged resources implement the `IDisposable` interface. `Dispose` is the only method in the interface and should be called when an instance of the type implementing the interface will not be used any more.

C# provides a special language construct to simplify the use of types implementing the `IDisposable` interface:

```
using (var reader = new StreamReader(filename))
{
    var contents = reader.ReadToEnd();
    Console.WriteLine($"Read {contents.Length} characters from
file.");
}
```

The instance declared in the `using` statement is only in scope inside the corresponding block of code. It is treated as a read-only variable: no other value can be assigned to it. As soon as the block of code gets executed, the `Dispose` method will get called on the declared instance. Even if an exception is thrown inside that block, the `Dispose` method will still be called, because the statement behavior is equivalent to the following code with a `try/finally` statement:

```
{
    var reader = new StreamReader(filename);
    try
    {
        var contents = reader.ReadToEnd();
        Console.WriteLine($"Read {contents.Length} characters from
file.");
    }
    finally
    {
        if (reader != null)
        {
            ((IDisposable)reader).Dispose();
        }
    }
}
```

```
    }  
  }  
}
```

There are three details in this code worth mentioning:

- The outer curly braces define a block of code which ensures that the declared variable gets out of scope immediately after it is disposed.
- In the finally block, the variable value is first checked for `null` to avoid a `NullReferenceException` being thrown.
- The instance is casted to `IDisposable` before invoking the `Dispose` method to handle the case when the method is implemented explicitly.

If more than one instance of the same type is declared in a single `using` statement, the `Dispose` method will be called on all of them:

```
using (StreamReader reader1 = new StreamReader(filename1),  
      reader2 = new StreamReader(filename2))  
{  
    // process the files  
}
```

The same approach doesn't work if the instances are of different types. In this case, multiple nested `using` statements must be used:

```
using (var reader1 = new StreamReader(filename1))  
using (var reader2 = XmlReader.Create(filename2))  
{  
    // process the files  
}
```

No curly braces need to be used for the outer `using` statement because even without them the inner `using` statement is nested inside it. Code formatting in Visual Studio will keep both `using` statements at the same indentation level to make the code more readable.

Due to its nature, the `using` statement can only be used when an instance of a disposable type needs to be accessible only within a single block of code. If that

instance will be used for a longer time, a different approach (as we will see shortly) must be taken to ensure that the Dispose method still gets called in the end.

Another common scenario is when a disposable type is used inside another type for the entire duration of the wrapping type lifetime. This requires the wrapping type to implement the IDisposable interface as well. When doing so, the standard basic dispose pattern should be followed to make the code comprehensible to others, and to avoid the risk of errors in the code:

```
public class DisposableClass : IDisposable
{
    bool disposed = false;
    private StreamReader reader;
    public DisposableClass(string filename)
    {
        reader = new StreamReader(filename);
    }

    public string Read()
    {
        if (disposed)
        {
            throw new ObjectDisposedException(nameof(DisposableClass));
        }
        return reader.ReadToEnd();
    }

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

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (reader != null)
            {
                reader.Dispose();
            }
        }
    }
}
```

```
        reader = null;
    }
}
disposed = true;
}
}
```

The pattern follows several best practices:

- The disposing logic is implemented in a separate `Dispose` method accepting a `bool` parameter. The parameterless `Dispose` method from the `IDisposable` interface always calls it with the value `true` so that the disposing logic gets executed. The value `false` is reserved for calls from the finalizer which should skip the disposing of any nested disposable instances because they could have already been finalized and shouldn't be accessed any more.
- The additional `Dispose` method accepting a `bool` parameter should be declared as `protected` and `virtual` to prevent the calls from outside the type and to allow deriving types to override it.
- The parameterless `Dispose` method should not be `virtual` to prevent deriving types from modifying its logic.
- No other overloads of the `Dispose` method should be implemented, so that the methods with this name are only used for the disposing logic.
- The parameterless `Dispose` method should call the `GC.SuppressFinalize` method *after* the `Dispose` method has successfully completed to indicate that the finalizer doesn't need to be called (see the following chapter for details).
- It should be safe to call the `Dispose` method multiple times.
- The `Dispose` method should never throw an exception unless the process is in a corrupt state and the application should close. Calls of the `Dispose` method from the `using` statement and from the finalizer do not handle any exceptions being thrown.
- If an instance of the type is attempted to be used after the `Dispose` method has been called, the `ObjectDisposedException` should be thrown whenever the nested disposable instance would need to be used.

- The basic Dispose pattern is designed to be used in two different scenarios:
- When unmanaged resources are used indirectly via nested disposable objects as in our example.
- When unmanaged resources are used directly, and a finalizer must be implemented as well.

In the first scenario, some practices required by the pattern are redundant (e.g. calling `GC.SuppressFinalize`). Despite that, the pattern should still be followed in its complete form to avoid incorrect behavior if a finalizer is added later.

31

Q31. How do Finalizers work?

Finalizers are required for types which directly interact with non-managed resources using native APIs. These types do not use a managed wrapper (in the form of another .NET type) which implements the `IDisposable` interface. Finalizers ensure that the unmanaged resource will be released by the garbage collector even if the `Dispose` method in that type is never invoked for a particular instance.

When the garbage collector runs and determines that an object has become inaccessible and can be released, it checks if it implements the `Finalize` method.

If it doesn't, it will be released immediately.

If it does, the garbage collector puts the object in its internal finalization queue structure and delays the releasing of memory until the next run.

The garbage collector will skip that step only if the `GC.SuppressFinalize` method was previously called for that instance (it should be called from its `Dispose` method). So, not only will failing to call the `Dispose` method on an instance delay the release of unmanaged resource until the next garbage collection, it will also postpone the

releasing of memory for at least one garbage collection run.

The `Finalize` method is meant only as a final resort and shouldn't be casually relied upon.

The `Finalize` method will be called on the instances in the finalization queue during the garbage collection run. The order in which the instances will be processed is not guaranteed, therefore no methods on other objects should be invoked from the `Finalize` method, including any nested disposable objects. The `Finalize` method will be invoked for each individual instance after it is processed by the garbage collector. No exceptions should be thrown in the finalizer as they will cause the process to be terminated.

In C#, the `Finalize` method must be implemented using the destructor syntax. It will automatically call the `Finalize` method in the base class if it exists. According to the basic `Dispose` pattern (see Chapter 30 for details), the finalization logic should be implemented in the `Dispose` method with a `bool` parameter:

```
public class ClassWithFinalizer : IDisposable
{
    bool disposed = false;
    IntPtr unsafeHandle;

    public ClassWithFinalizer(int size)
    {
        unsafeHandle = Marshal.AllocHGlobal(size);
    }

    ~ClassWithFinalizer()
    {
        Dispose(false);
    }

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

    protected virtual void Dispose(bool disposing)
```

```
{
    if (disposing)
    {
        // dispose nested disposable instances
    }
    if (unsafeHandle != IntPtr.Zero)
    {
        Marshal.FreeHGlobal(unsafeHandle);
        unsafeHandle = IntPtr.Zero;
    }
    disposed = true;
}
}
```

When the protected `Dispose` method is invoked from the finalizer (i.e. the destructor), `false` is passed as its argument so that invoking the `Dispose` method on nested disposable instances is skipped, and only directly obtained unmanaged resources are released. This is important because the nested instances might have already been processed by the garbage collector and shouldn't be accessed any more.

In the code sample, the `Marshal.AllocHGlobal` method is used in the constructor to allocate unmanaged memory. This memory is then released using `Marshal.FreeHGlobal` in the `Dispose` method.

Finalizers can be tricky to implement correctly and reliably, therefore it's a good idea to avoid them whenever possible. Often, the classes derived from the `SafeHandle` base class can be a safer alternative to manually handling unmanaged resources. They are also easier to implement correctly.

There are multiple different classes in the base class library derived from the `SafeHandle` class, each one supporting its own type of unmanaged resource, such as files (`SafeFileHandle`), registry keys (`SafeRegistryHandle`), pipes (`SafePipeHandle`), etc. They are all wrappers around handles for working with some type of resource which is returned from native APIs.

When using a `SafeHandle` derived class, there's no need to implement a finalizer. It's enough to only correctly implement the `IDisposable` interface:


```

public class ClassWithSafeHandle : IDisposable
{
    [DllImport("Kernel32.dll", SetLastError = true, CharSet =
    CharSet.Auto)]
    static extern IntPtr CreateFile(
    string fileName,
    [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,
    [MarshalAs(UnmanagedType.U4)] FileShare fileShare,
    IntPtr securityAttributes,
    [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,
    [MarshalAs(UnmanagedType.U4)] FileAttributes flagsAndAttributes,
    IntPtr template);

    bool disposed = false;
    SafeFileHandle safeHandle;

    public ClassWithSafeHandle(string filename)
    {
        var unsafeHandle = CreateFile(filename, FileAccess.Read,
        FileShare.Read, IntPtr.Zero, FileMode.Open, FileAttributes.
        Normal, IntPtr.Zero);
        safeHandle = new SafeFileHandle(unsafeHandle, true);
    }

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

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (safeHandle != null && !safeHandle.IsInvalid)
            {
                safeHandle.Dispose();
                safeHandle = null;
            }
        }
    }
}

```

```
    }  
    disposed = true;  
  }  
}
```

In this code sample, although I'm opening a file using a native API bypassing any managed code, I don't need to write a finalizer for the class. I delegate the responsibility to the `SafeFileHandle` class for correctly releasing the file handle returned by the native API. I only need to correctly dispose the class in the `Dispose` method. If the `Dispose` method of my class isn't called, then the `SafeFileHandle` class takes care of releasing the handle in its finalizer.

32

Q32. What is the purpose of Static class members?

In C#, any type member you create is by default an *instance member*. This means that it is tied to a particular instance of that type:

- Stateful members, such as properties and fields, can hold a different value for each instance.
- Stateless members, such as methods, will only have direct access to other members of the same instance. To gain access to members of other instances, these must be passed into the method as parameters.

This behavior is well demonstrated with the following sample code:

```
public class ClassWithInstanceMembers {  
    public int InstanceProperty { get; set; }  
    public void InstanceMethod() {  
        Console.WriteLine($"Property value: {InstanceProperty}");  
    }  
}
```

```
var instance1 = new ClassWithInstanceMembers();
instance1.InstanceProperty = 1;

var instance2 = new ClassWithInstanceMembers();
instance2.InstanceProperty = 2;

instance1.InstanceMethod(); // Property value: 1
instance2.InstanceMethod(); // Property value: 2
```

If we add the **static** modifier to a member definition, we tie it to the type instead. Static methods are typically used to implement factory and helper methods:

```
public class FibonacciNumberCalculator
{
    public static int GetFibonacciNumber(int n) {
        if (n < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(n));
        }
        else if (n < 2)
        {
            return n;
        }
        else
        {
            var numbers = new List<int> { 0, 1 };
            for (int i = 2; i <= n; i++)
            {
                numbers.Add(numbers[i - 1] + numbers[i - 2]);
            }
            return numbers[n];
        }
    }
}
```

They can be called by using the type name, instead of via a reference to the instance:

```
var fibonacci8 = FibonacciNumberCalculator.GetFibonacciNumber(8); //
21
```

To shorten the syntax for invoking them, C# 6 added the `using static` directive which makes all static members in a type accessible directly, without using any prefix. The type must be specified using a fully qualified name:

```
using static ClassNamespace.FibonacciNumberCalculator;
```

```
var fibonacci8 = GetFibonacciNumber(8); // 21
```

Inside static methods, the `this` keyword is not valid. Therefore, they cannot access instance members unless that instance is passed into them as a parameter. They can access other static members though.

For example, we can add a static field to our class to implement memoization, i.e. caching of previously calculated values so that we can return them immediately without having to recalculate them for every method call:

```
private static List<int> numbers = new List<int> { 0, 1 };
```

```
public static int GetFibonacciNumberWithMemoization(int n)
{
    if (n < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(n));
    }
    else if (n < numbers.Count)
    {
        return numbers[n];
    }
    else
    {
        for (int i = numbers.Count; i <= n; i++)
        {
            numbers.Add(numbers[i - 1] + numbers[i - 2]);
        }
        return numbers[n];
    }
}
```

Static members can only hold a single value per type. For generic types this means a single value per a given set of generic type arguments.

```
GenericClass<int>.StaticProperty = 1;
GenericClass<float>.StaticProperty = 2;

Console.WriteLine(GenericClass<int>.StaticProperty); // = 1
Console.WriteLine(GenericClass<float>.StaticProperty); // = 2
```

While this can be beneficial in many scenarios, it must also be carefully taken into consideration in multi-threaded applications. Since all threads access the same static members, race conditions can occur when multiple threads access the same member concurrently, resulting in inconsistent or invalid member value.

In the above example, two threads could calculate the same values in parallel and add them to the list. Because of duplicated values in the list, their indexes would be offset and the returned Fibonacci numbers would be incorrect.

The static modifier can also be used on a class:

```
public static class FibonacciNumberCalculator
{
    // ...
}
```

A static class can only contain static members and cannot be instantiated. It can't derive from other classes and other classes can't derive from it, i.e. it is automatically marked as **sealed**. An example of such a class in the .NET framework base class library is `System.Math`.

A special type of static members is Constructors. Unlike instance constructors, they can't be called explicitly and because of that they can't accept parameters. They will be executed automatically before any other static member of their type is accessed, as well as before any instance of that type is created by calling an instance constructor.

As such, static constructors are useful for initializing static members. In our second Fibonacci example we could for example initialize the memoization field with a static constructor instead of with the field initializer:

```
private static List<int> numbers;

static FibonacciNumberCalculator()
{
    numbers = new List<int> { 0, 1 };
}
```

In instance constructors, it's a best practice to do as little as possible and to avoid throwing exceptions. All of this is even more important for static constructors. They are only called once and if they fail, the type will remain uninitialized for the entire lifetime of the [application domain](#) which usually corresponds to the lifetime of the process.

Let's create a class with a static constructor, which can be configured to throw an exception:

```
public static class Config
{
    public static bool ThrowException { get; set; } = true;
}

public class FailingClass
{
    static FailingClass()
    {
        if (Config.ThrowException)
        {
            throw new InvalidOperationException();
        }
    }
}
```

It shouldn't come as a surprise that any attempt to create an instance of this class will result in an exception:

```
var instance = new FailingClass();
```

However, it won't be `InvalidOperationException`. The runtime will automatically wrap it into a `TypeInitializationException`. That's an important detail if you want to catch the

exception and recover from it.

```
try
{
    var failedInstance = new FailingClass();
}
catch (TypeInitializationException) { }
Config.ThrowException = false;
var instance = new FailingClass();
```

If the exception was thrown by an instance constructor, this code would catch that exception, change the configuration to avoid exceptions being thrown in future calls, and finally successfully create an instance of the class.

Since the static constructor for a class is only called once, it doesn't behave the same. If it throws an exception, then this exception will be rethrown whenever you want to create an instance or access the class in any other way. The class becomes effectively unusable until the application domain (or the process) is restarted. Because of this, having even a minuscule chance that the static constructor will throw an exception, is a very bad idea.

33

Q33. When are Extension Methods useful?

Extension methods are a special category of static methods which behave as if they were instance methods of the type they are extending. Unlike other static methods, they must always be placed inside a non-generic static class.

```
public static class StringExtensions
{
    public static bool ContainsIgnoreCase(this string instance,
    string substring)
    {
        return instance.IndexOf(substring, StringComparison.
        OrdinalIgnoreCase) >= 0;
    }
}
```

Their first parameter is crucial, making them extension methods instead of just ordinary static methods. It must always be declared with the `this` modifier. Its type specifies which type they are extending.

Extension methods are invoked with the same syntax as ordinary instance methods:

```
var input = "Hello, World!";  
var contains = input.ContainsIgnoreCase("world"); // true
```

The first parameter of the extension method will be set to the value of the instance they appear to have been called on.

At compile time, the compiler will determine the extension method which needs to be invoked. The generated code won't be any different from what it would be like if we invoked the extension method as a standard static method:

```
var input = "Hello, World!";  
var contains = StringExtensions.ContainsIgnoreCase(input, "world");  
// true
```

To make the extension method accessible, the namespace of the static class containing the extension method needs to be imported first with the **using** directive:

```
using ExtensionMethods;
```

To help with discoverability, Visual Studio displays extension methods next to instance methods in its IntelliSense popup windows. They are marked with a different icon to make them easily distinguishable from regular instance methods.

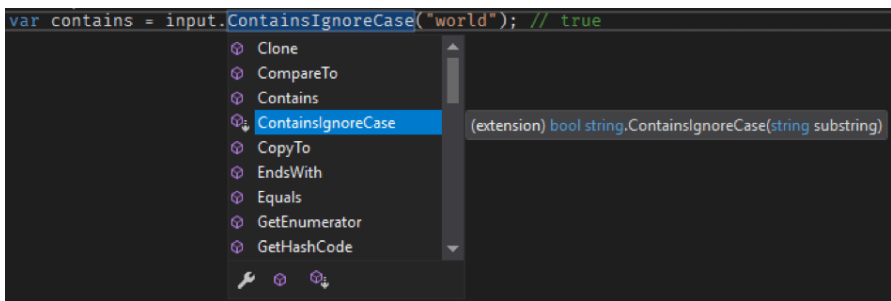


Figure 1: Extension method in Visual Studio's IntelliSense window

There are some restrictions to what extension methods can do in comparison to instance methods.

Since they are defined as static methods in a different class, they only have access

to public members of the class they are extending, just like any other method does. Private and protected methods of the extended class can't be accessed (at least not without resorting to reflection which can circumvent the accessibility restrictions). They are just syntactic sugar, nothing more.

Extension methods also can't override existing instance methods, even if they have a matching signature. Creating such a method is allowed:

```
public static bool Contains(this string instance, string substring)
{
    // use case insensitive comparison instead of case sensitive
    return instance.IndexOf(substring, StringComparison.
        OrdinalIgnoreCase) >= 0;
}
```

However, the compiler will always try to bind the called method to an instance method first. Only if such a method doesn't exist, it will start searching for a matching extension method. If we try to invoke the `Contains` method on a string variable after defining the above extension method, the instance method will still be called, i.e. case sensitive comparison will be used:

```
var input = "Hello, World!";
var contains = input.Contains("world"); // false
```

We can only call the extension method using the regular static method call syntax. Of course, we could also do that even if it was just a regular static method, not an extension method:

```
var input = "Hello, World!";
var contains = StringExtensions.Contains(input, "world"); // true
```

Extension methods can be used to seemingly add new instance methods even to types which we can't modify otherwise, because we don't have access to their code.

They are also very useful for interfaces which can't have methods of their own. Extension methods allow us to implement common code to be used with all types implementing a specific interface. This is extensively used in LINQ (Language INtegrated Query) which defines a large collection of extension methods for the `IEnumerable<T>` interface. They can be used with any type implementing this

interface:

```
var list = new List<int> { 1, 2, 3, 4, 5 };  
var countEven = list.Count(n => n % 2 == 0); // 2
```

Despite all the possibilities of extension methods, they should be used sparingly and only when equivalent functionality can't be achieved otherwise, i.e. by changing the class itself or deriving from it. Too many extension methods can make the code difficult to understand, especially to someone who is not familiar with all those extension methods.

If you were to use them, it's also more common to define extension methods in class libraries than in application code, so that they can be more widely reused.

34

Q34. What are auto-implemented properties and why are they better than public fields?

There are different ways how a class can allow consuming code to read values from and write values to it, without calling its methods.

The most bare-bones approach would be by exposing a public field:

```
public class SampleClass
{
    public int publicField = 42;
}
```

This provides no encapsulation whatsoever and should therefore *always* be avoided. The calling code can freely manipulate the field value:

```
var instance = new SampleClass();
var value = instance.publicField;
instance.publicField = -1;
```

The containing class can impose no control over it, apart from making the field **protected** or **private** and blocking any access to it from the outside. It could also make the field read-only but then even the containing class cannot modify it any more.

At the other end of the spectrum are properties which give the containing class full control over what the calling code is allowed to do. It can set different access modifiers for the getter and the setter. It can also execute custom code inside them, e.g. to validate the new value in the setter with a guard clause:

```
private int regularPropertyValue = 42;
public int RegularProperty
{
    get
    {
        return regularPropertyValue;
    }

    protected set
    {
        if (value < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(value));
        }
        regularPropertyValue = value;
    }
}
```

In this example, external code can only read the value of the property, but can't modify it:

```
var instance = new SampleClass();
var value = instance.RegularProperty;
instance.RegularProperty = -1; // won't compile
```

Derived classes can also set a new value to the property, but the base class can still validate the value before actually assigning it to the private backing field:

```
public class DerivedClass : SampleClass
{
```

```
public DerivedClass()
{
    RegularProperty = -1; // will throw exception
}
}
```

Auto-implemented properties are the middle ground between the two extremes.

They allow some of the encapsulation but have the advantage of much shorter code with less ceremony. The compiler generates a backing field for them, so there's no need to explicitly declare it in code.

The getter and the setter can still have different access modifiers:

```
public int AutoImplementedProperty { get; protected set; }
```

Just like with regular properties, external code can only read the value, but not modify it, while derived classes can modify the value as well.

Because the backing field is not accessible from code, it cannot be initialized directly. Before C# 6, it was only possible to initialize the value of auto-implemented properties in the constructor:

```
public SampleClass()
{
    AutoImplementedProperty = 42;
}
```

C# 6 introduced initializers for auto-implemented properties, so there's no need to create a constructor just to initialize the property:

```
public int AutoImplementedProperty { get; protected set; } = 42;
```

Although the backing field is not exposed and hence can't be marked as read-only directly, there's still a way to create immutable objects with auto-implemented properties. If only the setter is declared for an auto-implemented property, then the value of that property can only be set with the initializer syntax or in the constructor:

```
public int ReadOnlyAutoImplementedProperty { get; } = 42;
```

```
public SampleClass(int value)
{
    ReadOnlyAutoImplementedProperty = value;
}
```

In any other method, setting a value to this property is not allowed:

```
public void Method()
{
    ReadOnlyAutoImplementedProperty = -1; // won't compile
}
```

Such an auto-implemented property is truly immutable: it can be initialized when constructing the object but not modified at a later time.

The only real limitation of auto-implemented properties in comparison to regular properties is the inability to add custom logic to their getters and setters. If that's required, the full property syntax must be used.

However, existing auto-implemented properties can always be changed to regular properties later when such additional logic becomes necessary. This will not change the signature of that property or of the containing class. This allows the calling code to still use the class without any change. When the two classes are in different assemblies, it's not even necessary to recompile the calling code.

35

Q35. What are Expression-bodied members?

As the name implies, expression-bodied members are those members of a type which have a lambda expression as their body instead of a code block. Of course, this is only possible when all of the code in a member body can be rewritten as a single expression or statement. This will make the code shorter and in most cases, also easier to read.

The feature was first introduced in C# 6 when it could only be used for two types of members:

- For **methods**, their body could be rewritten as a lambda expression matching the return type of the method. For a method with a void return type, the lambda would be a statement not returning anything. The new syntax is particularly useful for short methods, such as `ToString` methods overrides:

```
public override string ToString() {  
    return $"{FullName}, {Age}";  
}
```

The same method could be rewritten with an expression body as follows:

```
public override string ToString() => $"{FullName}, {Age}";
```

- For **read-only properties**, the getter body could be converted to a lambda expression. For example, the following property in the old syntax:

```
public string FullName
{
    get
    {
        return $"{Name} {Surname}";
    }
}
```

could be compressed into a single line with the new syntax:

```
public string FullName => $"{Name} {Surname}";
```

In C# 7.0, the support for expression-bodied members has been expanded to other member types as well:

- **Properties with setters** can now have a lambda expression for both the getter and the setter. The latter often contains additional business logic otherwise the properties could be defined as auto-implemented properties:

```
private int age;
public int Age
{
    get
    {
        return age;
    }
    set
    {
        if (value > 0)
        {
            Age = value;
        }
        else

```

```

    {
        throw new ArgumentOutOfRangeException(nameof(value));
    }
}
}

```

Even such a setter can still be written in the lambda expression form:

```

private int age;
public int Age
{
    get => age;
    set => age = value > 0 ? value : throw new
        ArgumentOutOfRangeException(nameof(value));
}

```

The **if** statement must be replaced with a ternary operator because only a single assignment is allowed in an expression. The example also takes advantage of another feature that was introduced in C# 7.0: **throw expressions**. They allow us to throw exceptions from expressions and are often very useful in expression-bodied members.

- **Indexers** are very similar to regular properties, but they accept an additional parameter:

```

private Dictionary<string, int?> grades = new
    Dictionary<string, int?>();
public int? this[string course]
{
    get
    {
        if (grades.ContainsKey(course))
        {
            return grades[course];
        }
        else
        {
            return null;
        }
    }
}

```

```
    }  
    set  
    {  
        grades[course] = value;  
    }  
}
```

The conversion to the expression syntax is also similar:

```
private Dictionary<string, int?> grades = new  
Dictionary<string, int?>();  
public int? this[string course]  
{  
    get => grades.ContainsKey(course) ? grades[course] : null;  
    set => grades[course] = value;  
}
```

This time the ternary operator is used instead of the **if** statement in the getter.

- **Constructors** will often only assign parameter values to corresponding properties:

```
public Student(string name, string surname, int age)  
{  
    Name = name;  
    Surname = surname;  
    Age = age;  
}
```

To convert multiple assignments into a single expression, the **tuple** feature from C# 7.0 must be used:

```
public Student(string name, string surname, int age) => (Name,  
Surname, Age) = (name, surname, age);
```

The **deconstruction syntax** above specifies that each parameter from the tuple on the right will be assigned to the property at the same position in the tuple on the left.

- **Finalizers** also support the expression-bodied syntax. When they are implemented

using the standard pattern, they only contain a call to the `Dispose` method where the actual clean-up logic should be placed:

```
~Student()  
{  
    Dispose(false);  
}  
This makes them very suitable for conversion to expression-  
bodied members:  
~Student() => Dispose(false);
```

Although expression-bodied members typically involve less code than their equivalents with a code block as their body, this doesn't necessarily make the resulting code easier to read and understand.

When that's not the case, it's usually a better idea to use the longer syntax.

Struggling to understand the shorter code at a later time might take much longer than a few extra keystrokes would.

SECTION IV

STRING MANIPULATION

36

Q36. When can string interpolation be used instead of string formatting?

String interpolation is a more convenient alternative to composite formatting as implemented by the `string.Format`, `Console.WriteLine` and many other methods in the .NET framework base class library:

```
var name = "John";  
var formatted = string.Format("Hello, {0}!", name); // = "Hello,  
John!"
```

These methods compose the resulting formatted string by replacing the numbered placeholders in the format string passed as the first argument, with the values of the remaining arguments. The number in the placeholder indicates the position of the argument to be used. If no matching argument is provided, the code will still compile and a `FormatException` will be thrown at run time. There will be a warning displayed in Visual Studio though.

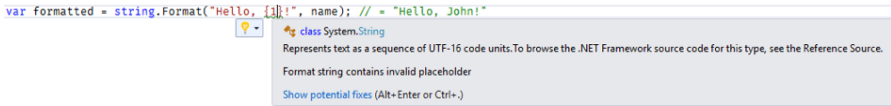


Figure 1: Invalid placeholder warning in Visual Studio 2017

Since C# 6, interpolated strings can be used instead:

```
var name = "John";
var formatted = $"Hello, {name}!"; // = "Hello, John!"
```

An interpolated string is indicated by the special \$ character in front of the opening quote. The values to be injected into the format string are specified with the expressions embedded directly inside it. This prevents potential errors caused by an incorrect index used in the format string for the `string.Format` and similar methods.

Although the expressions are inside the string, they are checked at compile time and Visual Studio provides full syntax highlighting and IntelliSense support:

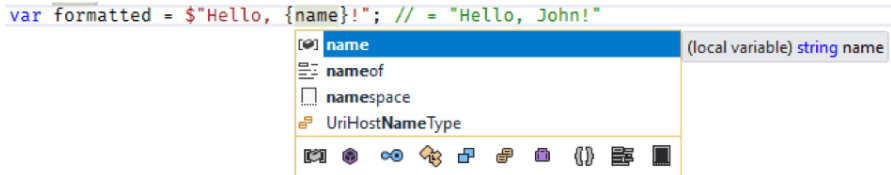


Figure 2: IntelliSense for the expression inside the interpolated string

Of course, any expression can be used, not only simple symbols from local scope:

```
var a = 4;
var b = 2;
var formatted = $"{a} + {b} = {a + b}"; // = "4 + 2 = 6"
```

Since braces denote embedded expressions, a double brace must be used when a brace should become a part of the resulting formatted string:

```
var A = new[] { 1, 2, 3, 4, 5 };
var formatted = $"{nameof(A)} = {{ {string.Join(", ", A)} }}"; // =
"A = { 1, 2, 3, 4, 5 }"
```

Other escape sequences, such as new line, can be used the same way as in ordinary strings:


```
var a = 3.5;
var b = 12.25;
var formatted = $"{a}\n+ {b}\n= {a + b}";
```

This will result in a multiline formatted string:

```
3.5
+ 12.25
= 15.75
```

In addition to regular string literals, C# also supports **verbatim string literals**. These don't require escape sequences for special characters (e.g. new line, tab, backslash), except for quotes ("" in place of "), and can span over multiple lines:

```
var regular = "File path:\r\n"C:\\Temp\"";
var verbatim = @"File path:
"C:\Temp\"";
```

Verbatim strings can also be used in combination with interpolated strings:

```
var a = 3.5;
var b = 12.25;
var formatted = $"{a}
+ {b}
= {a + b}";
```

The resulting formatted string will be the same as in the previous example.

Just like the composite formatting methods, interpolated strings support optional alignment and formatting components for each expression:

```
var a = 3.5;
var b = 12.25;
var formatted = $"{a,5:0.00}\n+ {b,5:0.00}\n= {a + b,5:0.00}";
```

The number after the comma specifies the width to which the expression value will be padded with spaces. Positive values will cause the value to be right-aligned inside that field and negative values will cause it to be left-aligned.

The string after the colon is the format string to be used with the value. Any standard or custom [formatting string](#) is supported.

The above interpolated string will generate a nicer formatted string:

```
3.50
+ 12.25
= 15.75
```

For each expression, both the alignment and the formatting component can be specified, only one of them, or neither:

```
var n = 3.5;
var alignmentAndFormatting = $"{n,5:0.00}"; // = " 3.50"
var alignmentOnly = $"{n,5}"; // = " 3.5"
var formattingOnly = $"{n:0.00}"; // = "3.50"
var neither = $"{n}"; // = "3.5"
```

Interpolated strings will be compiled into equivalent `Format.String` method calls. By default, the `CurrentCulture` set for the `CurrentThread` will be used.

To pass in a different culture, the correct overload of the `FormattableString.ToString` method must be used. To access it, the interpolated string must first be assigned to a variable of type `FormattableString` (The class is only available in .NET framework 4.6 and later.):

```
var culture = CultureInfo.GetCultureInfo("sl-SI");
var a = 3.5;
var b = 12.25;
FormattableString formattableString = $" {a,5:0.00}\n+ {b,5:0.00}\n= {a + b,5:0.00}";
var formatted = formattableString.ToString(culture); // = " 3,50\n+ 12,25\n= 15,75"
```

In most cases, interpolated strings are functionally equivalent to `string.Format` calls. However, there are a couple of exceptions.

For example, Entity Framework Core has special support for interpolated strings in the `IQueryable<T>.FromSql` method since version 2.0. One of its overloads accepts `FormattableString` as its parameter so that it intercepts the interpolated string before

it is formatted. This allows it to format the string differently, treating the expressions as query parameters and sanitizing them correctly:

```
using (var context = new EFContext())
{
    var param = "Doe";
    var query = context.Persons.FromSql($"SELECT * FROM Persons
WHERE LastName = {param};");
}
```

The above sample will result in a safe query being sent to the database:

```
SELECT * FROM Persons WHERE LastName = N'Doe';
```

There's still caution required if you're trying to take advantage of this functionality. Interpolated strings are very eagerly converted to regular strings by the compiler, therefore a different query will be generated if the interpolated string is first assigned to an implicitly typed variable. This will convert it to a string although it isn't explicitly assigned to a variable of type string:

```
using (var context = new EFContext())
{
    var param = "Doe";
    var sql = $"SELECT * FROM Persons WHERE LastName = {param};";
    var query = context.Persons.FromSql(sql);
}
```

In this case, the query sent to the database will most likely be invalid:

```
SELECT * FROM Persons WHERE LastName = Doe;
```

However, similar code can quickly appear to work but will be susceptible to SQL injection attacks:

```
var param = "Doe'; DROP TABLE Persons; --";
var sql = $"SELECT * FROM Persons WHERE LastName = '{param}'";
var query = context.Persons.FromSql(sql);
```

Another scenario in which interpolated strings can't be used instead of calls to `string.Format` is when they need to be translated. When using `string.Format`, the format

string can be read from the resource file with translations, instead of being a hard-coded literal in code:

```
var name = "John";  
// Properties.Strings.Localized = "Hello, {0}!"  
var formatted = string.Format(Properties.Strings.Localized, name);  
// = "Hello, John!"
```

The expressions embedded in interpolated strings are validated at compile time and can't be replaced at run time. This means that the interpolated string, including the expressions, (e.g. "Hello, {name}!") can't be replaced as a whole. Fortunately, the generated format string for an interpolated string (e.g. "Hello, {0}!") can be accessed at runtime if you use the `FormattableString` class. But replacing it with a translation (which is possible at run time) is inconvenient and negates most of the advantages of interpolated strings:

```
var name = "John";  
FormattableString formattableString = $"Hello, {name}!";  
// formattableString.Format = "Hello, {0}!"  
// formattableString.GetArguments() = [ name ]  
var formatted = string.Format(GetTranslation(formattableString.  
Format),  
    formattableString.GetArguments()); // = "Hello, John!"
```

As you can see, an explicit call to `string.Format` is required to use a different translated format string. I pass to it two values from the `FormattableString` representation of the interpolated string:

- The `Format` property returns the format string that can be used with methods implementing composite formatting (e.g. `string.Format`). Notice that it once again includes the positional placeholders, otherwise these methods couldn't use it.
- The `GetArguments()` method returns the array of arguments matching the positional placeholders in the format string.

Even with all this available, we still need to look up the translation for a specific interpolated string. With composite formatting this was easy: we could simply reference a key in the resource file by using a matching strongly typed property on the resource class (e.g. `Properties.Strings.Localized` in the first example).

There's no way to directly associate a resource key to an interpolated string. Instead we need to look up the translation based on the actual contents of the format string (e.g. "Hello, {0}" in the above example). This requires additional logic which would need to be implemented in the `GetTranslation` method used in the example:

```
private string GetTranslation(string original)
{
    // return the translation based on the content of the original
    string
}
```

Taking all of this into consideration, it's a better idea to keep using `string.Format` directly when the format strings must be localized.

37

Q37. How do locale settings affect string manipulation?

Many string operations can behave differently based on the locale settings which are in use. In the .NET framework, all culture-based string operations use the current culture which is set for the current thread (as the `CurrentCulture` property of the `Thread.CurrentCulture` static property) unless a different culture is specified for a specific call.

By default, the value of `CurrentCulture` property corresponds to the users' region and language settings in the operating system which is what the user typically expects. This makes it great in user-focused scenarios, i.e. when showing resulting strings to the user or parsing user input.

The fact that the behavior depends on the user settings makes these defaults unsuitable for internal string processing and programmatic communication between different components and systems.

In such scenarios, it is beneficial that the results are always and everywhere the same; deterministic and predictable. This can be achieved by changing the `CurrentCulture`

property on a thread if that thread never processes user-facing strings, or by specifying the desired culture for each operation.

The most obvious operations affected by culture settings, are formatting of dates and numeric values. When the `ToString` method is called on a value without a culture specified, the `CurrentCulture` is used (the samples in this chapter assume "en-US" locale settings are in use):

```
var date = new DateTime(2018, 9, 1);
var enDate = date.ToString("d"); // = "9/1/2018"
```

A different culture can be passed to the method to format the date differently:

```
var slCulture = CultureInfo.GetCultureInfo("sl-SI");
var slDate = date.ToString("d", slCulture); // = "1. 09. 2018"
```

The same approach can be used for numeric values. Although there is less variety in how the values are formatted, the differences are still important:

```
var pi = 3.14;
var enPi = pi.ToString(); // = "3.14"
var slPi = pi.ToString(slCulture); // = "3,14"
```

Of course, a fixed culture can also be specified when the `string.Format` method is used for composite formatting. The same culture will be used for formatting all the values injected into the format string:

```
var temperature = 21.5;
var timestamp = new DateTime(2018, 9, 1, 16, 15, 30);
var formatString = "Temperature: {0} °C at {1}";
var enFormatted = string.Format(formatString, temperature,
timestamp); // = "Temperature: 21.5 °C at 9/1/2018 4:15:30 PM"
var slFormatted = string.Format(slCulture, formatString,
temperature, timestamp); // = "Temperature: 21,5 °C at 1. 09. 2018
16:15:30"
```

Just like culture settings affect the formatting of dates and numeric values, they also affect the parsing of those values from a string. The same input string can be parsed very differently depending on what culture is in use:

```
var stringNumber = "3,14";  
Double.TryParse(stringNumber, out var enNumber); // = 314  
Double.TryParse(stringNumber, NumberStyles.Any, s1Culture, out  
var s1Number); // = 3.14
```

A less obvious operation affected by locale settings is string comparison:

```
var doubleS = "ss";  
var eszett = "ß";  
var equalOrdinal = string.Equals(doubleS, eszett); // = false  
var equalCulture = string.Equals(doubleS, eszett, StringComparison.  
CurrentCulture); // = true
```

The results of string comparison don't change only based on the culture. There are two additional important settings:

- Case sensitivity: are upper- and lower-case characters treated the same or not
- Ordinal or culture-based: ordinal comparison is based solely on the byte representation of the characters, while culture-based comparison adheres to linguistic rules.

Default string comparison in .NET is ordinal and case-sensitive. That's why the two different but equivalent ways to write the same character in German, are treated as different strings. When the strings are compared according to a culture, that linguistic rule is considered, and the strings are treated as equal even with non-German locale settings.

Ordinal case-insensitive string comparison is a special case which deserves additional explanation.

When strings are compared according to this rule, they are internally converted to upper case, using the invariant culture (a special culture based on the English language without any regional specifics). Then they are compared based on their byte representation, just like in the case of the default ordinal case-sensitive comparison.

String comparison settings are also important when sorting strings. Collections in the .NET framework base class library have an option to specify the sorting rules to use. Based on that setting, the strings will be sorted differently in the collection:


```
var czCulture = CultureInfo.GetCultureInfo("cs-CZ");
var words = new[] { "channel", "double" };
var ordinalSortedSet = new SortedSet<string>(words); // = {
"channel", "double" }
var cultureSortedSet = new SortedSet<string>(words, StringComparer.
Create(czCulture, ignoreCase: true)); // = { "double", "channel" }
```

As I used the Czech locale for the second collection, the order of the two strings was reversed in it. This happens because the Czech language treats the "ch" sequence as a different letter from C. It is positioned next to the letter H, i.e. several letters after the letter D, therefore it is considered "greater than" D.

There is another common string operation which can behave differently based on the culture in use: the change from lower- to upper-case letters or vice-versa:

```
var trCulture = CultureInfo.GetCultureInfo("tr-TR");
var lowerCaseI = "i";
var upperCaseIen = lowerCaseI.ToUpper(); // = "I"
var upperCaseItr = lowerCaseI.ToUpper(trCulture); // = "İ"
```

There aren't many languages where these rules are different, but that makes the bugs caused by them even more difficult to detect. Converting a string to upper or lower case is often used for case-insensitive string comparisons. If a fixed culture is not used for the operation, the results might be unexpected.

In the Turkish language, there are two different letters **I**: one with a dot and one without it. This property is preserved even when changing the case, i.e. the upper-case version of the "i" with a dot will still have a dot. That's different from the behavior in most other languages where the lower-case "i" has a dot and the upper-case one doesn't.

38

Q38. When can string manipulation performance suffer and how to avoid it?

In .NET, strings are immutable. This means that they can't be modified after they are created, even if it appears so after reading this piece of code:

```
var text = "a";  
text += "b";
```

When the above code is executed, the value of the text variable will be "ab". However, to store this value in memory, a *new* instance of the string class will be allocated. The old instance with value "a" will need to be garbage collected. However, the string code is highly optimized, and this behavior does not always negatively affect performance.

For example, multiple concatenations of literal strings in a single expression are processed during compile time. The following assignment wouldn't compile any differently even if a single string was used without concatenation:

```
var jsonLiteral = "{" +  
    "\"id\": 1," +
```

```

    "\"firstName\": \"John\", \" +
    "\"lastName\": \"Doe\", \" +
    "\"gender\": \"MALE\" \" +
    \"}";

```

Even when there are multiple concatenations in a single expression which can only be processed at runtime, the compiled code will still do only a single new allocation for the final result:

```

var numbers = new[] { "1", "2", "3" };
var concatenated = numbers[0] + numbers[1] + numbers[2];

```

Of course, if the number of values was dynamic, this approach wouldn't be possible. I'd have to use a loop instead:

```

var concatenated = string.Empty;
foreach (var number in numbers)
{
    concatenated += number;
}

```

For a small array, this still wouldn't cause any performance issues.

But for larger ones, you should start looking at alternative approaches. One option would be the **Concat** method which will concatenate all values in the array into a single string with a single memory allocation:

```

var concatenated = string.Concat(numbers);

```

If you wanted to insert separators between the values, use the `string.Join` method instead which is just as memory efficient:

```

var concatenated = string.Join(", ", numbers);

```

Of course, there are still scenarios which are too complex to solve using these methods. The following contrived code is one such example:

```
var concatenated = string.Empty;
foreach (var value in values)
{
    concatenated += value;
    concatenated += concatenated.Length;
}
```

Because of additional calculations inside the loop, you can't simply prepare the values for the concatenation in advance. As the number of iterations in the above block of code increases, the performance will start to suffer, since the garbage collector will have to release the memory allocations for the intermediate results.

In such cases, you should use the `StringBuilder` class instead. Unlike strings which are immutable, the `StringBuilder` class has a preallocated buffer. Inside the buffer, it performs all the string processing without additional memory allocations. The code won't be much different, but the performance will be much better when processing large arrays:

```
var stringBuilder = new StringBuilder();
foreach (var value in values)
{
    stringBuilder.Append(value);
    stringBuilder.Append(stringBuilder.Length);
}
var fromStringBuilder = stringBuilder.ToString();
```

I measured the execution time for the latest two examples on my machine with an array of 10,000 items and the difference was already noticeable:

```
String concatenation: 728 milliseconds
StringBuiler.Append: 2 milliseconds
```

The larger the dataset, the higher are the benefits of using the `StringBuilder` instead of concatenating strings. Still, there's no need to automatically replace all your string concatenation code with the `StringBuilder` class. If you know that you'll only be concatenating a few strings and don't need a loop for that, the additional complexity of the code isn't worth it.

When you go the `StringBuilder` route, you might be able to further optimize

performance. If you can estimate the size of the final string, you can use that knowledge to specify the `StringBuilder`'s capacity (i.e. the size of its buffer) in advance:

```
var stringBuilder = new StringBuilder(50000);
```

If you don't do this, the initial size will be 16 bytes. It will double whenever the string grows over that size. To avoid copying of data and garbage collection, additional capacity will be allocated as a separate chunk. The downside of this approach is that the memory won't be allocated contiguously which will affect the performance of some of the `StringBuffer`'s operations, such as `Insert`, `Replace` and `Remove`.

Another potential disadvantage of the `StringBuilder` class is its lack of support for search operations, such as `IndexOf` and `LastIndexOf`. To perform search, you first need to convert the `StringBuilder` instance to a string using the `ToString` method and then use the methods on the string class. If you need to do that often, you will lose most of `StringBuilder`'s benefits. In such scenarios you should test the performance of both approaches (using string concatenation and `StringBuilder`) to see which one is better for your requirement.

SECTION V

GENERIC AND COLLECTIONS

39

Q39. How can Generics improve code?

Generics were added in version 2.0 of both C# and the .NET framework. Before that, the only way to implement data structures which would work with different data types was to use the common base type: `object`. The `System.Collections` namespace still contains such collection classes from the era of .NET framework 1.0 and 1.1. They can be used with any type, however they also can't be restricted so that a single instance of a collection can only be used with items of a single type:

```
var list = new ArrayList();  
list.Add("one");  
list.Add(2); // will compile
```

When retrieving the stored items from the collection, they need to be cast to the right type before they can be fully used. Since items of different types can be stored in the same instance of the collection, we can cast the retrieved item to an incorrect type without any compile-time errors. Only at run-time, an exception will be thrown:

```
string first = (string)list[0];
```

```
string second = (string)list[1]; // throws InvalidCastException
```

To provide type safety, a special collection type can be used instead which restricts the type of items stored in it. Such collections are in the `System.Collections.Specialized` namespace:

```
var list = new StringCollection();  
list.Add("one");  
//list.Add(2); // will not compile  
  
string first = list[0]; // no cast necessary
```

This approach doesn't scale well, as a different collection type must be implemented for each type of item to store. These different collections will have a lot of similar or even identical code.

Generics solve these problems because:

- They allow code reuse, as the same collection type can be used to store any type of item.
- They provide type safety, as an instance is restricted to allow storing items of a single type only.

To achieve that, generic type definitions have generic type parameters (e.g. `List<T>` or `Dictionary<TKey, TValue>`). When creating an instance of a generic type, a generic type argument must be specified for each of the generic type parameters. This restricts the generic type so that it can only be used with the types given as the generic type arguments:

```
var list = new List<string>();  
list.Add("one");  
//list.Add(2); // will not compile  
  
string first = list[0]; // no cast necessary
```

Not only is the generic collection safer to use than its non-generic equivalent, it also performs better. I measured the execution time of the following two snippets of code on my machine:


```
var list = new ArrayList(values);
list.Sort();

var genericList = new List<int>(values);
genericList.Sort();
```

The second one consistently performed better:

```
ArrayList: 0.1142 ms
List<T>:    0.0110 ms
```

To improve performance, the Common Language Runtime (CLR) creates a specialized type at run time when a generic type is first instantiated with a specific set of generic type arguments:

- For value types, a new specialized type is created for each value type so that the values can be stored directly and there's no need for boxing to and unboxing from object. You can read more about this in chapter 13: "What is boxing and unboxing?".
- For reference types, a single specialized type is created for all different reference types because the references are always of the same size and the actual instances are stored on the heap.

However, generics are not limited to classes. Other types can also be generic, as well as methods too.

LINQ (Language Integrated Query) uses Generics a lot. The following single line of code is strongly dependent on generics:

```
var numbers = new[] { 1, 2, 3 };
var asStrings = numbers.Select(number => number.ToString()); // {
"1", "2", "3" }
```

The **Select** method is a generic method with two generic type parameters (TSource and TResult):

```
IEnumerable<TResult> Select<TSource, TResult>(this
IEnumerable<TSource> source, Func<TSource, TResult> selector);
```

It converts all items in the collection from one type to another. This is done by invoking the method argument typed as the following generic delegate for each item:

```
delegate TResult Func<in T, out TResult>(T arg);
```

Also, both the input and the output collections are generic interfaces (*IEnumerable<T>*).

In spite of all the generics involved, I didn't need to specify any generic type arguments explicitly. Thanks to type inference, the compiler could figure out all of them on its own based on the input collection and the lambda expression.

Considering all this, we can safely assume that LINQ would not even be possible without generics.

40

Q40. How to implement a Generic method or class?

Generic methods are not all that different from regular non-generic methods. The difference is that selected parameter types and return value types can be parameterized.

Often, you'll start with a non-generic method, and convert it to a generic one afterwards. Let's start with the following method, for example:

```
public static int CountDifferentInts(IEnumerable<int> numbers, int
compareTo)
{
    return numbers.Count(number => number != compareTo);
}
```

In its current implementation, the method will count how many numbers in the `ICollection<int>` value passed as the first argument are different from the `int` value passed as the second argument:

```
var numbers = new[] { -1, 0, 1, 0, -2, 0, 2 };
var different = CountDifferentInts(numbers, 0); // = 4
```

However, it will only work for ints. If we need the same functionality for a different data type, e.g. doubles, we will have to write another method.

Instead of simply reimplementing this method for doubles, we can convert it into a generic method. Then, we'll be able to use the same method for both int and double values.

We'll start by replacing `int` types in the parameters with a parameterized type `T`. This makes the method generic. To indicate that, the generic parameter `T` inside angle brackets immediately follows the method name:

```
public static int CountDifferentGeneric<T>(IEnumerable<T> vals, T
compareTo)
{
    // won't compile: Operator ,!= cannot be applied to operands of
    // type ,T and ,T
    return vals.Count(val => val != compareTo);
}
```

Notice that the type of the return value is still `int`. It's because we're returning a count of matching values which will be typed as `int` even if we're counting values of type `double`.

Unfortunately, the above method will not compile because the `!=` operand is not available for all types. Since the method must work, no matter what type we specify for `T`, we can't use this operator. Without changing the semantics of our method, we can replace the `!=` operator with the `object.Equals` static method which also tests for equality (and the `!` operator to negate the result, of course):

```
public static int CountDifferentGeneric<T>(IEnumerable<T> vals, T
compareTo)
{
    return vals.Count(val => !object.Equals(val, compareTo));
}
```

When invoking the method, we now need to specify a type for the `T` type parameter:

```
var numbers = new[] { -1, 0, 1, 0, -2, 0, 2 };
var different = CountDifferentGeneric<int>(numbers, 0); // = 4
```

In most cases, the compiler will be able to infer the correct type from the supplied values for the method parameters even if we don't explicitly specify the type. The method invocation syntax will then be identical to the invocation of a non-generic method:

```
var numbers = new[] { -1, 0, 1, 0, -2, 0, 2 };
var different = CountDifferentGeneric(numbers, 0); // = 4
```

But since the method is generic, we're not restricted to the `int` data type any more. The same method will also work with doubles:

```
var numbers = new[] { -1.5, 0, 1.5, 0, -2.5, 0, 2.5 };
var different = CountDifferentGeneric(numbers, 0d); // = 4
```

Since any type can be specified for the `T` type parameter, we can use the same method for non-numeric data types, as well:

```
var strings = new[] { "a", "", "b", "", "c", "", "d" };
var different = CountDifferentGeneric(strings, ""); // = 4
```

We just need to be aware of how the `Equals` method is defined for the type we use (the `object.Equals` static method calls the `Equals` instance method of its first argument).

Similar to generic methods, we can also define generic classes. Let's start with a non-generic class which expands on our sample method:

```
public class IntComparer
{
    private readonly int compareWith;

    public IntComparer(int compareWith)
    {
        this.compareWith = compareWith;
    }
}
```

```
public int CountDifferent(IEnumerable<int> numbers)
{
    return numbers.Count(number => number != compareWith);
}

public int CountEqual(IEnumerable<int> numbers)
{
    return numbers.Count(number => number == compareWith);
}
}
```

We have only moved the second parameter from the sample method to the class constructor and added a second method which checks the inverse condition.

Here's a small code snippet showing how this class can be used:

```
var numbers = new[] { -1, 0, 1, 0, -2, 0, 2 };
var instance = new IntComparer(0);
var different = instance.CountDifferent(numbers); // = 4
var equal = instance.CountEqual(numbers); // = 3
```

In a generic class, the type parameter is specified at the class level instead of the method level. Of course, it is then available for use in all members of the class:

```
public class GenericComparer<T>
{
    private readonly T compareWith;

    public GenericComparer(T compareWith)
    {
        this.compareWith = compareWith;
    }

    public int CountDifferent(IEnumerable<T> vals)
    {
        return vals.Count(val => !object.Equals(val, compareWith));
    }
}
```

```

public int CountEqual(IEnumerable<T> numbers)
{
    return numbers.Count(val => object.Equals(val, compareTo));
}
}

```

For the same reason as in the generic method, we had to replace the `!=` operator with the `Equals` method.

When instantiating the class, we now need to specify the type for the `T` type parameter. There's no support for inferring the type parameter for class constructors in C#, like there is for generic methods:

```

var numbers = new[] { -1, 0, 1, 0, -2, 0, 2 };
var instance = new GenericComparer<int>(0);
var different = instance.CountDifferent(numbers); // = 4
var equal = instance.CountEqual(numbers); // = 3

```

To avoid this limitation, instead of the constructor, it can sometimes be beneficial to create a generic factory method. It could be placed in a non-generic static class with the same name:

```

public static class GenericComparer
{
    public static GenericComparer<T> Create<T>(T compareTo)
    {
        return new GenericComparer<T>(compareTo);
    }
}

```

Because of type inference, there's no need to explicitly specify the generic type parameter anymore when invoking it:

```

var numbers = new[] { -1, 0, 1, 0, -2, 0, 2 };
var instance = GenericComparer.Create(0);
var different = instance.CountDifferent(numbers); // = 4
var equal = instance.CountEqual(numbers); // = 3

```

Of course, the rest of the code is the same as for the non-generic class. Again, we can substitute the `int` data type with a different one and the code will still work:

```
var numbers = new[] { -1.5, 0, 1.5, 0, -2.5, 0, 2.5 };  
var instance = GenericComparer.Create(0d);  
var different = instance.CountDifferent(numbers); // = 4  
var equal = instance.CountEqual(numbers); // = 3
```

Whenever we have code which is not really type specific and can be applied to any type, generics can help us avoid unnecessary code duplication and make maintenance and future changes simpler.

41

Q41. What are Generic type constraints?

By default, any type can be passed as a generic type argument. As a result, the code inside a generic method or class cannot assume much about its type parameter except for the fact that like every other .NET type, this parameter is a descendant of the **Object** type. So, only members of the **Object** type are available on variables of such unconstrained type parameters.

This is of course very restrictive and strongly limits what can be done inside a generic method or class. Additional methods can be made available by specifying constraints for the type parameters (using the **where** keyword):

```
public static bool IsAfter<T>(this T value, T compareTo) where T:
    IComparable
{
    return value.CompareTo(compareTo) > 0;
}
```

Constraints have the following two effects:

- When consuming generic methods or classes with constrained parameter types, only types which fulfill the constraints of a generic type parameter can be used in its place.
- The code inside the generic method or class can use additional members on variables of the constrained generic parameter type which are contained in the types provided as its constraints. It can also take advantage of other features resulting from these constraints (e.g. having a parameterless constructor available for the type) which are covered later in the chapter.

Probably the most commonly used type constraint is a base type or interface that the type must extend or implement, respectively:

```
public interface IPerson
{
    string FirstName { get; set; }
    string LastName { get; set; }
    int Age { get; set; }
}

public static IEnumerable<T> SelectOfAgeGeneric<T>(IEnumerable<T>
persons) where T: IPerson
{
    return persons.Where(person => person.Age >= 18);
}
```

The code inside this generic method can access the **Age** property of the IPerson interface. Consequentially, the consuming code is restricted to only using types which implement the IPerson interface:

```
var members = new List<Member>() { /* ... */ };
IEnumerable<Member> membersOfAge = SelectOfAgeGeneric(members);
```

You might wonder why is it better to have a generic method with the IPerson interface constraint instead of simply replacing the generic type parameter **T** with IPerson, as shown here:

```
public static IEnumerable<IPerson>
SelectOfAgeNonGeneric(IEnumerable<IPerson> persons)
{
```

```

    return persons.Where(person => person.Age >= 18);
}

```

The advantage can most easily be seen when we try to call such a non-generic method:

```

var members = new List<Member>() { /* ... */ };
IEnumerable<IPerson> membersOfAge = SelectOfAgeNonGeneric(members);

```

The return type of the non-generic method is always `IEnumerable<IPerson>`, even if a different type implementing the `IPerson` interface is used as the generic type parameter. This makes it impossible to use additional members of that type on the returned value without casting it back, and losing type safety in the process.

In contrast, the generic method with the `IPerson` interface constraint returns a value of the same type that was passed in as the parameter.

A very important aspect of the .NET type system is the difference between the value semantics of structs, and reference semantics of classes. Let's look at the following generic method for example:

```

public static void RenamePerson<T>(T person, string newName) where
T: IPerson
{
    person.FirstName = newName;
}

```

As long as we use a class for its type parameter, it will behave as expected. The value of the `FirstName` property will change:

```

var person = new PersonClass()
{
    FirstName = "John",
    LastName = "Doe",
    Age = 42
};
RenamePerson(person, "Jane");
Console.WriteLine(person.FirstName); // = "Jane"

```

However, there's nothing preventing the caller from passing in a struct instead of a class:

```
var person = new PersonStruct()
{
    FirstName = "John",
    LastName = "Doe",
    Age = 42
};
RenamePerson(person, "Jane");
Console.WriteLine(person.FirstName); // = "John"
```

Because of its value semantics, a copy of the struct will be accessed inside the generic method. Therefore, the change will not affect the original struct outside the method.

To prevent such incorrect usage of a method, a class constraint can be specified for the type parameter:

```
public static void RenamePerson<T>(T person, string newName)
where T: class, IPerson
{
    person.FirstName = newName;
}
```

Now, the code won't compile if a struct is used for the parameter type, instead of a class.

There's also a struct constraint available for the reverse case. Of course, both constraints can't be used for the same type parameter.

To be able to create an instance of the type parameter inside a generic method or class, the `new()` constraint must be used:

```
public static T CreatePerson<T>(string firstName, string lastName,
int age) where T: IPerson, new()
{
    var person = new T() // uses the constructor
    {
        // uses property setters from the interface
        FirstName = firstName,
```

```

        LastName = lastName,
        Age = age
    };
    return person;
}

```

This constraint specifies that the type must have a default (parameterless) constructor, making it available inside the generic method or class. The above method can now be used to create and initialize an instance of any type implementing the `IPerson` interface which has a default constructor:

```

var personStruct = CreatePerson<PersonStruct>("John", "Doe", 42);
var personClass = CreatePerson<PersonClass>("John", "Doe", 42);

```

Since all structs have a default constructor, using the `struct` constraint implies the `new()` constraint as well. Because of that, the `new()` constraint can't be used in combination with the `struct` constraint.

All constraints mentioned until now are available since C# 2.0 which is also the version when generics were introduced.

In C# 7.3, three additional type constraints have been added:

i) **The unmanaged type constraint** can only be matched by structs which are unmanaged (also called blittable). Such structs have the same representation in managed and unmanaged memory. They are value types which don't contain a reference type, either directly or indirectly. They may only contain basic types (numerical types including enums and pointers) and other unmanaged structs. The unmanaged type constraint can be used on generic helper functions which work with any unmanaged struct:

```

public static unsafe byte[] Serialize<T>(T value) where T :
unmanaged
{
    var bufferSize = sizeof(T);
    var buffer = new byte[bufferSize];
    fixed (byte* bufferPtr = &buffer[0])
    {
        Buffer.MemoryCopy(&value, bufferPtr, bufferSize, bufferSize);
    }
}

```

```
    return buffer;
}
```

ii) **The System.Delegate type constraint** makes it possible to write generic helper methods for handling delegates, such as the following extension method for type-safe combining of delegates (you can read more about combining delegates in Chapter 29: "How do delegates and events differ?"):

```
public static TDelegate Combine<TDelegate>(this TDelegate source,
TDelegate target) where TDelegate : Delegate
{
    // Delegate.Combine must be used instead of the + operator
    return (TDelegate)Delegate.Combine(source, target);
}
```

iii) Similarly, **the Enum type constraint** enables generic helper methods for enum types. One of the use cases could be a common type-safe extension method for extracting a value from an attribute applied to an enum value:

```
public enum FuelType
{
    Petrol,
    Diesel,
    [Description("Liquid petroleum gas")]
    LPG
}

public static string GetDescription<TEnum>(this TEnum value) where
TEnum : Enum
{
    var type = typeof(TEnum);
    var member = type.GetMember(value.ToString());
    var attribute = (DescriptionAttribute)Attribute.
        GetCustomAttribute(member[0], typeof(DescriptionAttribute));
    return attribute != null ? attribute.Description : value.
        ToString();
}
```

42

Q42. What is Covariance and Contravariance?

With generic types, usually expressions can only be assigned to variables when the type arguments on the left-hand side and the right-hand side of the assignment match:

```
ICollection<string> list = new List<string>();  
ICollection<object> lessDerivedList = list; // won't compile
```

Attempting to assign an expression of a generic type with a more derived type argument (`string` in the example above) to a variable of the same generic type with a less derived type argument (`object` in the example above) will result in compilation error.

This property of generic types is called **invariance**.

Since C# 4.0 and the corresponding .NET framework 4.0, this is not true for all generic types any more. For example, the `IEnumerable<T>` generic interface does not have such a restriction:

```
IEnumerable<string> strings = new List<string>();  
IEnumerable<object> objects = strings;
```

The `IEnumerable<string>` type with the more derived `string` type argument can be assigned to the `IEnumerable<object>` type with the less derived `object` type argument without losing type safety.

This property is called **covariance**.

It is only allowed when the type parameter is always in the role of a return value and never in the role of a parameter, as is the case with `IEnumerable<T>` and `IEnumerator<T>` (the `out` keyword indicates covariance and will be explained in more detail later in the chapter):

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

Type safety is not compromised because the code expecting an instance of `IEnumerable<object>` will only be able to access members of the `object` type on any item retrieved from it. The instance of `IEnumerable<string>` will provide items of the `string` type instead of the `object` type. Since the `string` type is derived from the `object` type, it also includes all the members of the `object` type as expected by the code.

The `ICollection<T>` generic interface is not covariant because the generic type parameter is also in the role of method parameters, e.g.:

```
void Add(T item);
```

A type implementing the `ICollection<string>` interface will expect a parameter of the `string` type and will be allowed to access any of its members. If we would be allowed to assign an expression of the `ICollection<string>` type to a variable of the `ICollection<object>` type, we could then invoke the `Add` method with an argument of any type (because all types derive from `object`). Since the `object` type does not include all the members of the `string` type, this would break type safety. When the type implementing the `ICollection<string>` interface attempts to access a member defined in the `string` type on the newly added item which is of a different type, that call would fail.

However, method parameters allow the reverse.

If a method in a generic type expects a parameter of the `object` type, we can safely pass it an instance of the more derived `string` type. There are generic types which allow this, e.g. the `Predicate<T>` generic delegate:

```
Predicate<object> objectPredicate = obj => obj.ToString().Length >
5;
Predicate<string> stringPredicate = objectPredicate;
```

The `Predicate<object>` type with the less derived `object` type argument can be assigned to the `Predicate<string>` type with the more derived `string` type argument without losing type safety.

This property is called **contravariance**.

It might seem counterintuitive, but it is allowed because the type parameter is only in the role of a method parameter and not in the role of the return value:

```
public delegate bool Predicate<in T>(T obj);
```

Because we have assigned a delegate of the `Predicate<object>` type to a variable of the `Predicate<string>` type, that predicate will receive a value of the `string` type instead of the `object` type. As already explained, this will not compromise type safety.

Covariance and contravariance are properties of type parameters, not of generic types. Therefore, a single generic type can have multiple type parameters, with each one of them being covariant, contravariant or neither (i.e. invariant). For example, the following generic delegate has one contravariant and one covariant type parameter:

```
public delegate TResult Func<in T, out TResult>(T arg);
```

The `T` type parameter is in the role of the method parameter and is contravariant. The `TResult` type parameter is in the role of the return value and is covariant. This makes the following code valid:

```
Func<object, string> originalDelegate = obj => obj.ToString();
Func<string, object> variantDelegate = originalDelegate;
```

We have instantiated a delegate which expects an argument of the `object` type and

returns a result of the **string** type. When we assign it to a variable of the `Func<string, object>` type, we will have to pass in an argument of the **string** type (which will have all the expected members of the **object** type). We will also expect to receive a result of the object type but will get a value of the string type instead (which will again have all the expected members of the object type).

In the Base Class Library (BCL), there are many more generic types with covariant and contravariant type parameters, but we are not restricted to those. The generic types we declare can also have covariant and contravariant type parameters. However, these are restricted to interface and delegate types.

The following generic interface includes all the different parameter types (covariant, contravariant, and invariant):

```
public interface IVariant<out TCovariant, in TContravariant,
    TInvariant>
{
    TCovariant Covariant();
    void Contravariant(TContravariant input);
    TInvariant Invariant(TInvariant input);
}
```

The keyword used indicates the role in which the type parameter can appear:

- Covariant type parameters can only be in the role of return values. They are declared with the **out** keyword.
- Contravariant type parameters can only be in the role of method parameters. They are declared with the **in** keyword.
- Invariant type parameters can be in any role. They are declared without a keyword.

The same rules apply to declaring generic delegate types:

```
public delegate TCovariant VariantDelegate<out TCovariant, in
    TContravariant>(TContravariant input);
```

The covariant type parameter with the **out** keyword is in the role of the return value. The contravariant type parameter with the **in** keyword is in the role of the input

parameter. Although there are no invariant type parameters in this delegate, we could declare one without a keyword and use it both in the role of the return value and the input parameter.

43

Q43. How does the compiler handle the IEnumerable<T> interface?

The generic IEnumerable<T> interface provides a unified way for iterating over a collection of items. It is implemented by all the built-in collections in the Base Class Library.

The interface contains only a single method GetEnumerator which returns an instance of the IEnumerator<T> interface. The members of this interface can then be used to implement the iteration:

```
using (var enumerator = collection.GetEnumerator())
{
    while (enumerator.MoveNext())
    {
        var item = enumerator.Current;
        // process item
    }
}
```

As you can see, only two members of the IEnumerator<T> interface are explicitly used in the code:

- The **Current** property returns the item at the current position in the collection. In the generic version of the interface, it is typed as **T**.
- The **MoveNext** method must be called to move to the next item in the collection.
- Additionally, the **Dispose** method is implicitly called by the using statement.

When the `IEnumerator<T>` instance is returned by the `GetEnumerator` method of the `IEnumerable<T>` interface, the value of the **Current** property is undefined. Before accessing it, the `MoveNext` method must be called at least once to set its value to the first item in the collection. The `MoveNext` method returns **true** as long as there is still an item at the new position. It returns **false** if the end of the collection was reached. At that point, the `Current` property should not be accessed anymore because its value becomes undefined again.

There's no way to move backwards in the collection. To start once again from the beginning, a new instance of the `IEnumerator<T>` interface must be obtained by calling the `GetEnumerator` method of the `IEnumerable<T>` interface again.

The collection must not be modified while an iteration is in progress. Doing so will invalidate all existing instances of the `IEnumerator<T>` interface for that collection.

Since iterating over collections is a very common operation, there's a special construct in the C# language for doing that. Instead of manually accessing members of the `IEnumerator<T>` interface, the **foreach** statement can be used to achieve the same functionality:

```
foreach (var item in collection)
{
    // process item
}
```

Using the `foreach` statement is recommended over directly interacting with the `IEnumerable<T>` and the `IEnumerator<T>` interfaces. In both cases, the IL generated by the C# compiler will almost be equivalent. The only difference is that the iteration variable is read-only when using the `foreach` statement. You are not allowed to modify its value inside the loop.

It's also worth mentioning that the `foreach` statement doesn't require the instance it iterates over to implement the `IEnumerable<T>` interface (or its non-generic counterpart `IEnumerable`). It's enough that it has a suitable method named

GetEnumerator like the class in the following convoluted example:

```
public class NoIEnumerable<T>
{
    private readonly IEnumerable<T> enumerable;

    public NoIEnumerable(IEnumerable<T> enumerable)
    {
        this.enumerable = enumerable;
    }

    public IEnumerator<T> GetEnumerator() => enumerable.
    GetEnumerator();
}
```

The return type doesn't need to be `IEnumerator<T>` (or `IEnumerator`) either. The only requirement is that it has the `Current` property and the `MoveNext` method with the correct signature.

If we wrap any collection into the class above, we can still iterate over it with the `foreach` statement:

```
var noEnumerable = new NoIEnumerable<int>(collection);

foreach (var item in noEnumerable)
{
    // process item
}
```

Although this isn't something we need to think about when using the `foreach` statement, it is still good to know. For example, this is the reason why we can iterate over instances of the `Span<T>` struct which was recently introduced:

```
var array = new[] { 1, 2, 3, 4, 5 }; // initialize array
Span<int> span = array; // implicit cast
var subSpan = span.Slice(start: 1, length: 3); // reference part of
the array
```

```
foreach (var item in subSpan)
{
    // process item
}
```

Since `Span<T>` is a so-called ref struct type, it can't implement interfaces. But it does have the `GetEnumerator` method which allows it to be used with the `foreach` statement as one would intuitively expect.

You can read more about `Span<T>` and the ref struct types in general in the upcoming Chapter 84 – "How was support for passing values by reference expanded in C# 7?".

44

Q44. How to implement a method returning an IEnumerable?

Although the IEnumerable interface contains only a single method, that method returns an implementation of the IEnumerator interface. This means that both the IEnumerable and the IEnumerator interfaces will need to be developed at the same time. A majority of development effort will need to be put in the latter.

The class implementing the IEnumerator interface is often closely coupled to the matching class implementing the IEnumerable interface. Therefore, it's a good practice to implement the IEnumerator interface as a private class nested inside the class implementing the IEnumerable interface. This prevents it from being instantiated on its own outside the parent class:

```
public class ArrayWrapper<T> : IEnumerable<T>
{
    private readonly T[] array;

    public ArrayWrapper(T[] array)
    {
        if (array == null)
        {
```



```

        throw new ArgumentNullException(nameof(array));
    }
    this.array = array;
}
public IEnumerator<T> GetEnumerator() => new
ArrayEnumerator(array);
IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
private class ArrayEnumerator: IEnumerator<T>
{
    private readonly T[] array;
    public ArrayEnumerator(T[] array)
    {
        this.array = array;
    }

    // ...
}
}

```

The class in this piece of code we just saw will reimplement the `IEnumerable` interface over an array only to demonstrate how this could be done. There's no practical need for doing that in production code because the array already implements the interface itself.

You will notice that I implemented both the generic and the non-generic version of the `GetEnumerator` method. This is required by the `IEnumerable<T>` interface so that the `IEnumerable` interface is also always implemented.

The same requirement is imposed by the `IEnumerator<T>` interface. To also implement the `IEnumerator` interface, both the generic and non-generic versions of the `Current` property must be implemented. Other members of both interfaces have matching signatures, so this is the only additional requirement.

The key part of a class implementing the `IEnumerator` interface is tracking the current position in the collection being iterated over:

```

private int index = -1;

public T Current => index >= 0 && index < array.Length ?
array[index] : default(T);

```

```
object IEnumerator.Current => Current;
public bool MoveNext()
{
    index++;
    return index < array.Length;
}
```

In this implementation, the **Current** property returns the default value for the type when the enumerator is positioned before the first item or after the last item. This is not required by the interface specification as the value is not defined in such cases and the Current property should not even be accessed from consuming code.

There are two more methods that need to be implemented:

```
public void Reset()
{
    index = -1;
}
public void Dispose()
{ }
```

The **Reset** method resets the position back to the beginning. However, it's not required to be implemented and should not be relied upon in the consuming code. The method could simply throw a `NotSupportedException` when invoked.

The **Dispose** method is empty in my case because the class isn't handling any unmanaged resources. Otherwise I would need to do all the cleanup inside it (e.g. close a connection to a database). You can read more about implementing the Dispose method correctly in Chapter 30: "What's special about the `IDisposable` interface?".

With all this code in place, a method returning an `IEnumerable` can now simply create a new instance of the `ArrayWrapper<T>` class:

```
public static IEnumerable<T> GetArrayWrapper<T>(T[] array) => new
ArrayWrapper<T>(array);
```

Fortunately, you usually won't need to implement the `IEnumerable` and `IEnumerator` interfaces to return an `IEnumerable`. You can avoid writing all of that code by taking advantage of support for iterators in C#. By using the **yield return** keywords,

equivalent functionality can be achieved with much lesser code:

```
public static IEnumerable<T> GetArrayWrapperWithYield<T>(T[] array)
{
    if (array == null)
    {
        throw new ArgumentNullException(nameof(array));
    }
    for (var index = 0; index < array.Length; index++)
    {
        yield return array[index];
    }
}
```

Apart from the `null` check, the method consists only of a simple loop, iterating over the items in the array and returning each one using the `yield return` statement. Every time this line is reached, the execution of the method will stop until the `MoveNext` method of the `IEnumerator` instance in use is called again. The compiler will automatically generate the code required to achieve such functionality. The generated code will actually be very similar to the hand coded implementation of the `IEnumerable` interface we just saw.

The `yield return` syntax is not only useful for iterating over collections stored in memory, it could just as well be used for retrieving data from external data sources involving I/O operations, e.g. files or databases.

The values returned could also be calculated on-the-fly as in the following example:

```
public static IEnumerable<long> GetFibonacciSequence()
{
    yield return 1;
    yield return 1;
    long preprevious = 1;
    long previous = 1;
    while (true)
    {
        var current = preprevious + previous;
        preprevious = previous;
```

```
        previous = current;  
        yield return current;  
    }  
}
```

The `IEnumerable` returned by this method will return the Fibonacci sequence of numbers, when iterated over (i.e. each number will be the sum of the previous two numbers). Since all the values are not stored in memory, but calculated as needed, the numbers could in theory be iterated over forever (if we ignore the problem of numeric overflow when the calculated value goes over the maximum value supported by the `long` data type).

While such an implementation might seem like a good idea, it can cause problems for consuming code which is not aware of the implementation details. When the `IEnumerable` returned by this method is used in a simple `foreach` loop, it will never exit. The same will happen if the `ToArray` or `ToList` LINQ method is called on it. It would be much safer if the `GetFibonacciSequence` method accepted a parameter with the stopping criteria for the loop, e.g. a maximum number of items or a maximum value returned.

45

Q45. What advantages do Collection classes have over Arrays?

Arrays are represented in memory as a contiguous block of memory with individual values in it stored in subsequent memory locations.



Figure 1: Memory representation of an array

This allows efficient access to individual values since the memory location of each value in the array can easily be calculated from the index:

```
var array = new[] { 1, 2, 3, 4, 5 };  
var item = array[2];
```

The [time complexity](#) of the operation is $O(1)$. Changing a value at a specific index is just as efficient:

```
array[2] = 0;
```

On the other hand, inserting an additional value into the array can't be implemented efficiently and has the time complexity $O(n)$:

```
public static T[] InsertIntoArray<T>(T[] array, int index, T item)
{
    var newArray = new T[array.Length + 1];
    for (int i = 0; i < index; i++)
    {
        newArray[i] = array[i];
    }
    newArray[index] = item;
    for (int i = index + 1; i < newArray.Length; i++)
    {
        newArray[i] = array[i - 1];
    }
    return newArray;
}

var newArray = InsertIntoArray(array, 2, 0);
```

Because all of the allocated memory is already filled with values, a new larger block of memory needs to be allocated. Existing values must be copied to it and the new value must be inserted in the correct location. Since a new array has to be created, any code that previously used the array still has a reference pointing to the original array without the new value, and must update it to point to the new array with all the items.

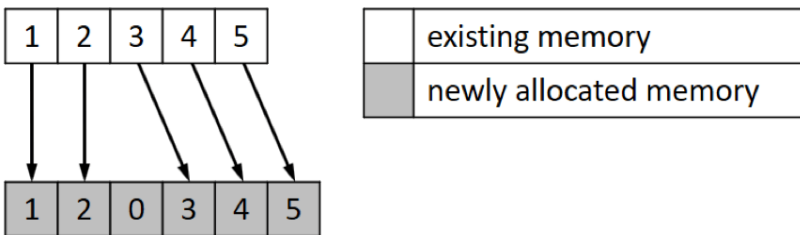


Figure 2: Inserting a value into an array

Although the `List<T>` collection also has its values stored in a contiguous block of memory, it resolves at least some of these inconveniences:

- It hides the actual memory representation from the consumer. Even if a new block of memory needs to be allocated, it can still be accessed through the unchanged reference to the same instance of `List<T>`.
- The class implements methods for additional common operations (such as adding and removing values) which can be used without having to know the internal details of how the values are stored in memory:

```
var list = new List<int> { 1, 2, 3, 4, 5 };
list.Insert(2, 0);
```

To avoid new memory allocations whenever an item is inserted into the list, a larger buffer of memory is allocated in advance. As long as some of it is still unused, the insert operation can avoid copying all values and only move the values positioned after the inserted value to make place for it:

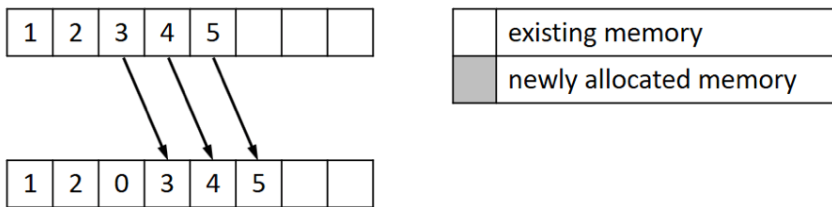


Figure 3: Inserting a value into a list

The Time complexity of the Insert operation is still $O(n)$, but at least no memory operations mean less work for the garbage collector.

Adding an item to the end of the list will have the time complexity $O(1)$ in most cases:

```
list.Add(6);
```

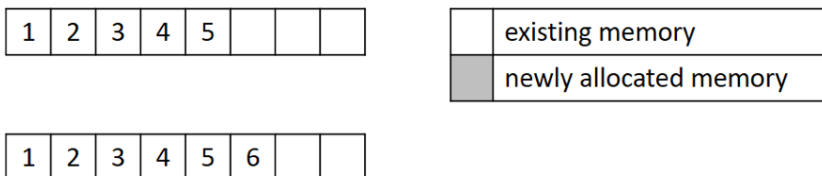


Figure 4: Adding an item to the end of the list

Only when all the allocated memory is already in use, the **Add** operation will have the complexity $O(n)$ because all items will have to be copied to the new memory buffer:

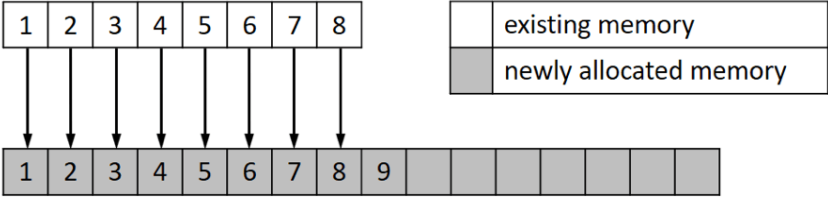


Figure 5: Adding an item to the end of the list when the buffer is full

The memory buffer will double in size whenever this happens. When we can anticipate the required capacity in advance, we can specify it when creating the instance of `List<T>` and avoid memory reallocations altogether:

```
var list = new List<int>(25);
```

There are other types of collections available in the Base Class Library. If we don't need to access the items by index but would like to insert items more efficiently, we can use `LinkedList<T>` instead of `List<T>`:

```
var linkedList = new LinkedList<int>(new [] { 1, 2, 3, 4, 5 });
var secondNode = linkedList.First.Next;
linkedList.AddAfter(secondNode, 0);
```

As the name implies, the values are stored in a doubly linked list:

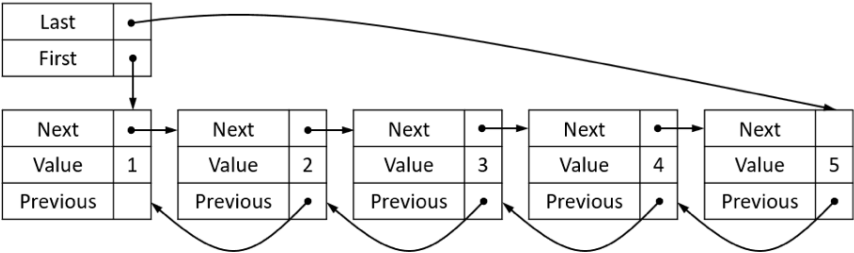


Figure 6: Memory representation of a linked list

To insert a new value anywhere in the list, only the references from its two neighboring nodes need to be updated. This results in the time complexity $O(1)$:

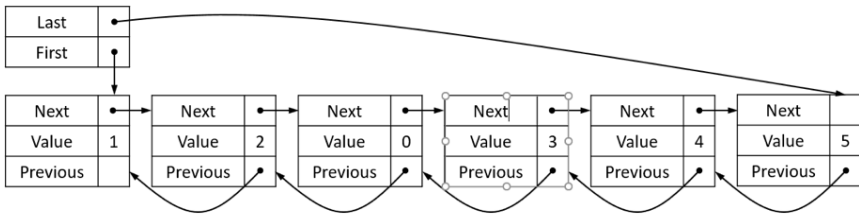


Figure 7: Linked list after inserting a new value

While iterating through a linked list in either direction is just as efficient as in the list, accessing an item at a specific index has the time complexity $O(n)$ instead of $O(1)$, because it can't be accessed directly. The list must be iterated through, until the requested index is reached.

Another important aspect of collections in the .NET framework is that they implement interfaces describing the operations they support. Therefore, a lot of code can be written without explicitly specifying a particular implementation of the interface:

```
ICollection<int> collection = new List<int> { 1, 2, 3, 4, 5 };
```

```
foreach (var item in collection)
{
    // process items
}
```

This makes it easy to replace one implementation of the interface with another one that better matches the requirements (performance or otherwise). Only an instance of a different type needs to be assigned to the variable and nothing else needs to be changed in the code:

```
ICollection<int> collection = new LinkedList<int>(new[] { 1, 2, 3,
4, 5 });
```

```
foreach (var item in collection)
{
    // process items
}
```

There are many different implementations of collection interfaces [in the Base Class](#)

[Library](#), differing in their memory representation or other implementation details. Additional implementations are provided by third party libraries distributed through NuGet (such as [FSharp.Collections](#), [OptimizedPriorityQueue](#) and [JB.Common.Collections](#)). If necessary, you can even create your own implementation matching your specific requirements. None of that is possible if the code is using arrays directly.

46

Q46. Why are there so many Collection classes and which one should I use?

The Base Class Library contains many different collection classes. Most of them are placed in the `System.Collections` and the `System.Collections.Generic` namespaces, but there are quite a few present in other namespaces too. This can make it difficult to decide when to use which collection.

In most cases, only collections in the `System.Collections.Generic` namespace should be considered.

The collections in `System.Collections` are all non-generic, originating from the time before Generics was added to C# with the release of the .NET framework 2.0 and C# 2.0. Except when maintaining legacy code, they should be avoided because they don't provide type safety, and when used with value types, display poor performance compared to their generic counterparts.

The collections in other namespaces are highly specialized for specific scenarios and are usually not applicable to code from other problem domains. E.g. the `System.Dynamic.ExpandoObject` class implements multiple collection interfaces but is primarily meant for implementing dynamic binding (you can read more about dynamic binding in Chapter 21 – “How can dynamic binding be implemented in C#?”).

Even when only looking at generic collections in the `System.Collections.Generic` namespace, the choice can still seem overwhelming.

Fortunately, the collections can easily be grouped into a few categories based on the interfaces they implement. These determine which operations are supported by a collection and consequently in which scenarios they can be used.

The common interface for collections is the `ICollection<T>` interface. It inherits from the `IEnumerable<T>` interface which provides the means for iterating through a collection of items:

```
IEnumerable<int> enumerable = new List<int>() { 1, 2, 3, 4, 5 };

foreach(var item in enumerable)
{
    // process items
}
```

The `ICollection<T>` interface has the `Count` property and methods for modifying the collection:

```
ICollection<int> collection = new List<int>() { 1, 2, 3, 4, 5 };

var count = collection.Count;
collection.Add(6);
collection.Remove(2);
collection.Clear();
```

These members should suffice for implementing a simple collection. Three interfaces extend this base interface in different ways to provide additional functionalities.

The `IList<T>` interface describes collections with items which can be accessed by their index. For this purpose, it adds an indexer for getting and setting a value:

```
IList<int> list = new List<int> { 1, 2, 3, 4, 5 };

var item = list[2]; // item = 3
list[2] = 6; // list = { 1, 2, 6, 4, 5 }
```

It also includes methods for inserting and removing a value at a specific index:

```
list.Insert(2, 0); // list = { 1, 2, 0, 6, 4, 5 }
list.RemoveAt(2); // list = { 1, 2, 6, 4, 5 }
```

The most commonly used class implementing the `ICollection<T>` interface is `List<T>`. It will work great in most scenarios, but there are other implementations available for more specific requirements. For example, the `SynchronizedCollection<T>` class has a built-in synchronization object which can be used to make it thread-safe.

The `ISet<T>` interface describes a set, i.e. a collection of unique items that doesn't guarantee to preserve their order. It also provides an **Add** method for adding individual items to the collection:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };

var added = set.Add(0);
```

However, its **Add** method will only add the item to the collection if it's not already present in it. The return value will indicate whether the item was added or not.

The rest of the [methods in the interface](#) implement different standard set operations from [set theory](#). The following methods modify the collection:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };

set.UnionWith(new[] { 5, 6 }); // set = { 1, 2, 3, 4, 5, 6 }
set.IntersectWith(new[] { 3, 4, 5, 6, 7 }); // set = { 3, 4, 5, 6 }
set.ExceptWith(new[] { 6, 7 }); // set = { 3, 4, 5 }
set.SymmetricExceptWith(new[] { 4, 5, 6 }); // set = { 3, 6 }
```

The others only perform tests on the collection and return a Boolean value:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };

var isSubset = set.IsSubsetOf(new[] { 1, 2, 3, 4, 5 }); // = true
var isProperSubset = set.IsProperSubsetOf(new[] { 1, 2, 3, 4, 5 });
// = false
var isSuperset = set.IsSupersetOf(new[] { 1, 2, 3, 4, 5 }); // =
```

```

true
var isProperSuperset = set.IsProperSupersetOf(new[] { 1, 2, 3, 4, 5
}); // = false
var equals = set.SetEquals(new[] { 1, 2, 3, 4, 5 }); // = true
var overlaps = set.Overlaps(new[] { 5, 6 }); // = true

```

The most basic implementation of `ISet<T>` is the `HashSet<T>` class. If you want the items in the set to be sorted, you can use `SortedSet<T>` instead. Immutable versions of both are available as well.

`IDictionary<TKey, TValue>` is the third interface extending the `ICollection<T>` interface. In contrast to the previous two, this one is for storing key-value pairs instead of standalone values, i.e. it uses `KeyValuePair<TKey, TValue>` as the generic parameter in `ICollection<T>`. Its members are designed accordingly. The indexer allows getting and setting the values based on a key, instead of an index:

```

IDictionary<string, string> dictionary = new Dictionary<string,
string>
{
    ["one"] = "ena",
    ["two"] = "dva",
    ["three"] = "tri",
    ["four"] = "štiri",
    ["five"] = "pet"
};

var value = dictionary["two"];
dictionary["zero"] = "nič";

```

The **Add** method can be used instead of the indexer to add a pair to the collection. Unlike the indexer, it will throw an exception if the key is already in the collection.

```
dictionary.Add("zero", "nič");
```

Of course, there are also methods for checking if a key is in the collection and for removing an existing pair based on its key value:

```

var contains = dictionary.ContainsKey("two");
var removed = dictionary.Remove("two");

```

The latter will return a value indicating whether the key was removed or not. If not, it is because it wasn't in the collection in the first place.

There are also properties available for retrieving separate collections of keys and values:

```
ICollection<string> keys = dictionary.Keys;
ICollection<string> values = dictionary.Values;
```

For scenarios with no special requirements, the `Dictionary<TKey, TValue>` class is the go-to implementation of the `IDictionary<TKey, TValue>` interface. Two different implementations are available if keys need to be sorted (`SortedDictionary<TKey, TValue>` and `SortedList<TKey, TValue>`), [each with its own advantages and disadvantages](#). Many more implementations of the interface are available in the Base Class Library.

There are two collection classes worth mentioning which unlike the ones we discussed so far don't implement `ICollection<T>` or any specialized interface derived from it.

The first one is the `Queue<T>` class which implements a [FIFO \(first in, first out\)](#) collection. Only a single item in it is directly accessible, i.e. the one that's in it for the longest time. The methods for adding and removing an item have standard names for such a data structure:

```
var queue = new Queue<int>(new[] { 1, 2, 3, 4, 5 });

queue.Enqueue(6);
var dequeuedItem = queue.Dequeue();
```

There's also the `Peek` method which returns the same item as the `Dequeue` method but leaves it in the collection:

```
var peekedItem = queue.Peek();
```

The second one is the `Stack<T>` class which is similar to `Queue<T>`, but it implements a [LIFO \(last in, first out\)](#) collection. The single item that's available in this collection is the one that was most recently added. The methods are named accordingly:

```
var stack = new Stack<int>(new[] { 1, 2, 3, 4, 5 });
```

```
stack.Push(6);  
var poppedItem = stack.Pop();  
var peekedItem = stack.Peek();
```

Although this overview doesn't cover all the available collection classes, it should serve as a very good guide for choosing the right collection class for the most common scenarios.

47

Q47. What problem do Concurrent Collections solve?

The regular generic classes in the Base Class Library have one very important deficiency if you need to use them in a multi-threaded application: they are not thread-safe.

At least, not fully.

While most of them support several concurrent reads, no concurrent write access is allowed. So, even the reading operations can't be considered thread-safe because there could always be consumers performing write operations.

As soon as the collection needs to be modified, any access to it from multiple threads must be synchronized. The simplest approach to implementing such synchronization involves using the `lock` statement with a common synchronization object:

```
public class DictionaryWithLock<TKey, TValue>
{
    private readonly object syncObject;
    private readonly Dictionary<TKey, TValue> dictionary;
```

```
public DictionaryWithLock()
{
    syncObject = new object();
    dictionary = new Dictionary<TKey, TValue>();
}

public TValue this[TKey key]
{
    get
    {
        lock (syncObject)
        {
            return dictionary[key];
        }
    }
    set
    {
        lock (syncObject)
        {
            dictionary[key] = value;
        }
    }
}

public void Add(TKey key, TValue value)
{
    lock (syncObject)
    {
        dictionary.Add(key, value);
    }
}
```

Although this approach would work, it is far from optimal.

All access to the inner dictionary is fully serialized, i.e. the next operation can only start when the previous is completed. Since the collection allows multiple concurrent readers, performance can be significantly improved if it takes that into account and only restricts concurrent access for writing operations.

Fortunately, there's no need for implementing this functionality from scratch. The Base Class Library comes with the `ReaderWriterLockSlim` class which can be used to implement this specific functionality in a relatively simple manner:

```
public class DictionaryWithReaderWriterLock<TKey, TValue>
{
    private readonly ReaderWriterLockSlim dictionaryLock = new
    ReaderWriterLockSlim();
    private readonly Dictionary<TKey, TValue> dictionary;

    public DictionaryWithReaderWriterLock()
    {
        dictionaryLock = new ReaderWriterLockSlim();
        dictionary = new Dictionary<TKey, TValue>();
    }

    public TValue this[TKey key]
    {
        get
        {
            dictionaryLock.EnterReadLock();
            try
            {
                return dictionary[key];
            }
            finally
            {
                dictionaryLock.ExitReadLock();
            }
        }
        set
        {
            dictionaryLock.EnterWriteLock();
            try
            {
                dictionary[key] = value;
            }
            finally
            {
                dictionaryLock.ExitWriteLock();
            }
        }
    }
}
```

```
        dictionaryLock.ExitWriteLock();
    }
}

public void Add(TKey key, TValue value)
{
    dictionaryLock.EnterWriteLock();
    try
    {
        dictionary.Add(key, value);
    }
    finally
    {
        dictionaryLock.ExitWriteLock();
    }
}
```

Depending on the type of operation being synchronized, there are two different lock modes used: reading or writing. The class will allow any number of concurrent read operations. On the other hand, the write lock will be exclusive and won't allow any other read or write access at the same time.

Even with the use of synchronization primitives provided by C# and the .NET framework, writing production-quality synchronization code is no small feat. Both of my wrapper classes implement synchronization for only a small subset of operations. You would want to implement at least the full `IDictionary<TKey, TValue>` interface if not all the members of the wrapped `Dictionary<TKey, TValue>` class.

That's somewhat similar to what the concurrent collections in the `System.Collections.Concurrent` namespace do. They provide thread-safe implementations of collection interfaces. However, thread-safety imposes certain limitations:

- Concurrent collections are less performant in single-threaded scenarios than their non-concurrent counterparts because of the synchronization code overhead. You should therefore only use them when you require thread safety.
- Except for `ConcurrentDictionary<TKey, TValue>` which implements the `IDictionary<TKey, TValue>` interface, concurrent collections can't directly

replace similar non-concurrent ones. The `ConcurrentBag<T>` class is the closest alternative to `List<T>` and `HashSet<T>`. And even the `ConcurrentQueue<T>` and the `ConcurrentStack<T>` classes don't provide the same methods as their non-concurrent counterparts `Queue<T>` and `Stack<T>` despite the similar names.

The concurrent collection classes can be safely used in multi-threaded applications. They have extra members in addition to those from the implemented collection interfaces which can prove useful in multi-threaded scenarios: `TryAdd`, `TryGetValue`, `TryPeek`, `TryRemove`, `TryPop`, `TryTake` etc.

Still, there are minor caveats when using these collections. For example, the `ConcurrentDictionary<TKey, TValue>` class includes methods which are not fully [atomic](#). The overloads of `GetOrAdd` and `AddOrUpdate` methods which accept delegates as parameters will invoke these delegates outside the synchronization locks.

Let's inspect the following line of code to understand the implications of this:

```
var resource = concurrentDictionary.GetOrAdd(newKey, key =>
ExpensiveResource.Create(key));
```

Such method calls are common when the dictionary is used as a cache for instances of classes which might be expensive to create, but can safely be used from multiple threads. Without knowing the details of how the `ConcurrentDictionary<TKey, TValue>` is implemented, one would assume that this line of code will either return an instance from the cache or create a single new instance using the provided delegate when there's no matching instance yet in the dictionary. It should then return that instance on all subsequent requests.

However, since the delegate is invoked outside the synchronization lock, it could be invoked from multiple different threads which reached this line of code when the key was not yet in the dictionary.

Although only one of the created instances will be stored in the dictionary in the end, the fact that multiple instances were created could be an issue. At the very least, it will affect performance if the creation takes a long time.

If the factory method should only ever be called once for a specific key, you will need to write additional synchronization code yourself. The simplest approach (but not the most performant one) would be to prevent creating multiple resources at the same time by using a `lock` statement, no matter the key:

```
var success = concurrentDictionary.TryGetValue(newKey, out var
resource);
if (!success)
{
    lock (syncObject)
    {
        // was the resource created by another thread after the check
        // outside of lock?
        if (!concurrentDictionary.TryGetValue(newKey, out resource))
        {
            resource = ExpensiveResource.Create(newKey);
            concurrentDictionary.TryAdd(newKey, resource);
        }
    }
}
```

It is important to always thoroughly read the documentation for all concurrent collection classes to be aware of such implementation details.

48

Q48. How are Immutable Collections different from other collections?

Immutable collections aren't included in the Base Class Library. To use them, the `System.Collections.Immutable` NuGet package must be installed in the project.

Immutable collections can't be changed after they are created. This automatically makes them safe to use in multi-threaded scenarios without any synchronization code since there's no way for another thread to modify them and make their internal state inconsistent. You can read about other advantages of immutable types in a forthcoming chapter – "What functional programming features are supported in C#?".

This fundamental design decision of course affects the API of immutable collection classes. They don't have public constructors. There are still two ways to create a new instance, though:

- A regular collection can be converted into an immutable one using an extension method:

```
var immutableList = new[] { 1, 2, 3, 4, 5 }.ToImmutableList();
```

- An empty collection which is exposed as a static property can be modified by

adding new items to it:

```
var immutableList = ImmutableList<int>.Empty.AddRange(new[] {  
    1, 2, 3, 4, 5 });
```

All operations which would normally modify the collection, return a new instance instead. It's important to remember that the returned value must be used from there on, instead of the original instance:

```
immutableList = immutableList.Add(6);
```

The fact that each method returns a new instance of the same class makes it easy to chain multiple method calls:

```
immutableList = immutableList  
    .Add(6)  
    .Add(7)  
    .Add(8);
```

However, this approach should be avoided whenever possible because each method call in such a chain creates a new instance of the collection. Although the immutable collections are implemented in a way to reuse as much of the original collection as possible when creating a new instance, some memory allocations are still required. This means more work for the garbage collector.

The best way to minimize this problem is to use methods which can perform the required modifications in a single call. The above chain of method calls could for example be replaced with the following single method call:

```
immutableList = immutableList.AddRange(new[] { 6, 7, 8 });
```

More complex operations can be performed with a single method call. By following the naive approach from mutable collections, the following block of code would be used to remove all even numbers from a list:


```
for (var i = immutableList.Count - 1; i >= 0; i--)
{
    if (immutableList[i] % 2 == 0)
    {
        immutableList = immutableList.RemoveAt(i);
    }
}
```

To improve performance, the following equivalent line of code should be used instead:

```
immutableList = immutableList.RemoveAll(n => n % 2 == 0);
```

However, specialized methods are not available for all types of complex modifications. For example, there's no single method available to both add and remove specific items as below:

```
immutableList = immutableList
    .Add(6)
    .Remove(2);
```

In such cases a builder can be used instead:

```
var builder = immutableList.ToBuilder();
builder.Add(6);
builder.Remove(2);
immutableList = builder.ToImmutable();
```

The **ToBuilder** method will create a builder for an immutable collection which is mutable and implements the interface of the corresponding mutable collection. Its internal memory structure makes it efficient to convert back to an immutable collection, but the operations will modify the same instance instead of always creating a new one. Only when calling the **ToImmutable** method, will an immutable instance be created again in a very efficient manner. This will reduce the amount of work for the garbage collector as much as possible.

So, when should immutable collections be used instead of regular or concurrent mutable ones?

A typical use case is of multi-threaded applications, especially when a thread doesn't

need to have access to the latest state of the collection and can use a potentially stale immutable instance which was originally passed to it. When that's the case, immutable collections might offer better performance in spite of additional work for the garbage collector because there are no synchronization locks required.

If an application needs to undo the operations on collections, it might make sense to use immutable collections even in single-threaded scenarios. Snapshots of previous states are a side product of immutable collections and don't require any additional resources to create. They can for example be stored in a stack and used to easily undo the last change:

```
snapshots.Push(immutableList);  
immutableList = immutableList.Add(6);  
  
immutableList = snapshots.Pop(); // undo: revert list to previous  
state
```

To achieve the same functionality with mutable collections, copies must be created before each modification:

```
snapshots.Push(list.ToList());  
list.Add(6);  
  
list = snapshots.Pop();
```

Not only is creating copies time consuming, the copies also require more memory than their immutable counterparts. The copies of mutable collections don't share any memory between them, unlike immutable collections which often share large parts when they are created through modification. The exact extent depends on the type of the collection, e.g. there's no sharing between `ImmutableArrays`, but `ImmutableLists` share parts of the tree structure which is used to store data internally.

SECTION VI

LANGUAGE INTEGRATED QUERY (LINQ)

49

Q49. What is the basic structure of a LINQ query?

LINQ (i.e. Language INtegrated Query) is a feature which was introduced in .NET framework 3.5. An important part of it is a special syntax inside C# for writing queries against a data source in a strongly typed manner.

It can be used against different types of data sources (e.g. .NET collections, relational databases, XML documents), abstracting away the individual specialized query languages for each data source. This removes the need for embedding literal strings in your code with those queries, e.g. SQL (Structured Query Language) queries for databases or XPath queries for XML documents.

If you're familiar with SQL, the structure of a LINQ query will seem very familiar. A simple SQL query consists of four main parts:

```
SELECT FirstName, LastName -- projection
FROM persons -- data source
WHERE Age > 21 -- filter
ORDER BY Age DESC; -- sorting
```

A LINQ query has exactly the same parts, but their ordering is different.

It always starts with the specification of the data source:

```
var query =
    from person in persons
    select person;
```

The **persons** variable in the above expression must be already defined at this point, e.g.:

```
private readonly Person[] persons = new[]
{
    new Person("John", "Doe", 33),
    new Person("Jane", "Doe", 42),
    new Person("Johnny", "Doe", 13)
};
```

The **person** variable is declared within the expression and it plays the role of an iteration variable inside the query. As the collection is iterated through, it will have the value of the currently processed **Person** instance.

The select part of the query is always at the end. In its current form, it will return the full **Person** instance.

When executed (e.g. iterated through), this query will simply return the full contents of the data source:

```
FirstName: John, LastName: Doe, Age: 33
FirstName: Jane, LastName: Doe, Age: 42
FirstName: Johnny, LastName: Doe, Age: 13
```

This result can be printed out with the following code:

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

To return only some of the properties of the source class (i.e. **Person**), an anonymous type can be created:

```
var query =  
    from person in persons  
    select new { person.FirstName, person.LastName };
```

Only the properties listed will now be included. They will have the same name as in the source class:

```
FirstName: John, LastName: Doe  
FirstName: Jane, LastName: Doe  
FirstName: Johnny, LastName: Doe
```

Filtering of the data source needs to be done using the **where** keyword:

```
var query =  
    from person in persons  
    where person.Age > 21  
    select new { person.FirstName, person.LastName };
```

The iteration variable can be accessed in the expression following the **where** keyword. The query will only return those instances for which this filter expression evaluates to true. For our sample data, the result will be:

```
FirstName: John, LastName: Doe  
FirstName: Jane, LastName: Doe
```

The final part of a basic LINQ query specifies the sorting:

```
var query =  
    from person in persons  
    where person.Age > 21  
    orderby person.Age descending  
    select new { person.FirstName, person.LastName };
```

Again, the iteration variable is accessible. The items in the query result will be sorted (descending in our example) based on the value the **orderby** expression evaluates to:

```
FirstName: Jane, LastName: Doe  
FirstName: John, LastName: Doe
```

If you compare the final LINQ query with the equivalent SQL query mentioned at the beginning, you can see a high level of similarity. The most important difference is that the **select** part is at the end of the LINQ query and at the beginning of the SQL query. The rest are just minor syntax differences.

However, when used with C# and the .NET framework, the LINQ query is strongly typed and validated at compile time against the types that are being queried.

If a property of the **Person** class in our example was renamed or removed, the code wouldn't compile any more. As long as you keep the types in sync with the database structure (using either [migrations](#) for code-first modelling or [scaffolding](#) for DB-first modelling), any errors in queries will be detected at compile time.

The literal string with the SQL query on the other hand is not validated against the underlying database structure. A mismatch can only be detected at runtime.

50

Q50. How is LINQ query syntax translated to C# method calls?

LINQ query syntax is much more similar to SQL queries when compared to regular C# code.

Although this might not be obvious at first sight, it still compiles down to regular method calls. They are all implemented as extension methods for the `IEnumerable<T>` and `IQueryable<T>` interface.

Let's look at how individual query parts are translated to corresponding method calls.

Projection is implemented by the `Select` method. The mapping is defined with a delegate, usually implemented as a lambda function:

```
var query = persons.Select(person => new { person.FirstName,
person.LastName });
```

The above call is equivalent to the following query syntax:

```
var query =
    from person in persons
```



```
select new { person.FirstName, person.LastName };
```

Each extension method returns a value of type `IEnumerable<T>`, which makes it possible to chain multiple method calls. Filtering can be added with a call to the `Where` method:

```
var query = persons
    .Where(person => person.Age > 21)
    .Select(person => new { person.FirstName, person.LastName });
```

The `OrderBy` and `OrderByDescending` methods can be used to define the sort order:

```
var query = persons
    .Where(person => person.Age > 21)
    .OrderByDescending(person => person.Age)
    .Select(person => new { person.FirstName, person.LastName });
```

In all the examples so far, the order of the method calls matches the order of the corresponding parts of equivalent code in query syntax:

```
var query =
    from person in persons
    where person.Age > 21
    orderby person.Age descending
    select new { person.FirstName, person.LastName };
```

Unlike the query syntax, the method syntax doesn't strictly require the `Select` method to be the last one in the query. The code will still work even if we put it elsewhere:

```
var query = persons
    .Select(person => new { person.FirstName, person.LastName,
        person.Age })
    .OrderByDescending(person => person.Age)
    .Where(person => person.Age > 21);
```

However, the different order affects the semantics because the methods execute in the order they are listed. In the latest example, I included the `Age` property in the anonymous type returned by the `Select` method. That's required because the methods placed after the `Select` method use the `Age` property.

If I excluded the **Age** property from the anonymous type, the code wouldn't compile anymore because subsequent method calls couldn't access that property:

```
var query = persons
    .Select(person => new { person.FirstName, person.LastName })
    .OrderByDescending(person => person.Age) // won't compile
    .Where(person => person.Age > 21);
```

Also, when using method calls, the **Select** method is of course not required at all if the unchanged source type is to be returned. In the query syntax, such a query must still include the **select** part:

```
var query =
    from person in persons
    where person.Age > 21
    select person;
```

Direct conversion to method calls would result in the following code:

```
var query = persons
    .Where(person => person.Age > 21)
    .Select(person => person);
```

But since this **Select** call simply returns each item unmodified, it can be omitted without affecting the result:

```
var query = persons.Where(person => person.Age > 21) ;
```

Each LINQ syntax has its own advantages and disadvantages:

- The query syntax is shorter and often easier to understand.
- The method syntax is standard C# code. This makes it easier to see what operations will be performed.

However, not all of the LINQ API is available through the query syntax. Many methods must be invoked directly because they don't have an equivalent in the query syntax.

For example, the **Take** method can be called to limit the maximum number of items returned by a query:

```
var query = persons
    .Where(person => person.Age > 21)
    .OrderByDescending(person => person.Age)
    .Take(1)
    .Select(person => new { person.FirstName, person.LastName });
```

The **Skip** method is often used in combination with the **Take** method. It causes a specified number of items at the beginning of the sequence to be skipped. Together with the **Take** method, it can be used to implement pagination:

```
var query = persons
    .Where(person => person.Age > 21)
    .OrderByDescending(person => person.Age)
    .Skip(10)
    .Take(10)
    .Select(person => new { person.FirstName, person.LastName });
```

With 10 items per page, the above example would return the results for the second page (it would skip the first 10 items and return the second 10).

Although there is no equivalent keyword to be used in the query syntax, it is possible to combine the query syntax with method calls:

```
var query = (
    from person in persons
    where person.Age > 21
    orderby person.Age descending
    select new { person.FirstName, person.LastName }
).Skip(10).Take(10);
```

There are other commonly useful LINQ operations that are only available as extension methods.

- The **Any** method returns true when at least one item in the collection matches the given predicate:

```
var any = persons.Any(person => person.Age > 21);
```

When used without a predicate, the **Any** method will return true if the collection

contains any items at all, i.e. if it is non-empty:

```
var notEmpty = persons.Any();
```

- In contrast, the **All** method only returns true when every item in the collection matches the given predicate:

```
var all = persons.All(person => person.Age > 21);
```

- When dealing with nested collections, the **SelectMany** method can be used to flatten the results from multiple inner collections into a single collection. Without it, two nested loops must be used to iterate through all the items in the inner collections:

```
var guests = rooms.Select(room => room.Guests);
```

```
foreach (var roomGuests in guests)
{
    foreach (var guest in roomGuests)
    {
        // process guest
    }
}
```

By using the **SelectMany** method instead of the **Select** method, a single loop is enough to achieve the same:

```
var guests = rooms.SelectMany(room => room.Guests);
```

```
foreach (var guest in guests)
{
    // process guest
}
```

This simplifies the code when we don't care about the individual parent items. It also allows us to use further LINQ methods on the flattened collection:

```
var guestsOfAge = rooms
    .SelectMany(room => room.Guests)
    .Where(guest => guest.Age > 21);
```

When such method-only LINQ operations are required, everyone can decide for oneself whether to still partially use the query syntax, or to write the entire query using the method syntax.

51

Q51. How to express Inner and Outer Joins using LINQ?

The terms inner and outer join originate from relational databases. They are both used to combine data from two separate data sources, into a single result.

The inner join includes only those items in the result from the first data source which have a match in the second data source. It includes them once for each match.

Id	FirstName	LastName	StudentId	CourseId	DateRegistered
1	John	Doe	1	101	2018-02-15
2	Jane	Roe	1	102	2018-02-15
3	Max	Lane	2	102	2018-02-17

Id	FirstName	LastName	CourseId	DateRegistered
1	John	Doe	101	2018-02-15
1	John	Doe	102	2018-02-15
2	Jane	Roe	102	2018-02-17

Figure 1: Inner join

To create a LINQ query with an inner join, we will start with the following two collections:

```
private readonly Student[] students = new[]
{
    new Student { Id = 1, FirstName = "John", LastName = "Doe" },
    new Student { Id = 2, FirstName = "Jane", LastName = "Roe" },
    new Student { Id = 3, FirstName = "Max", LastName = "Lane" }
};

private readonly Registration[] registrations = new[]
{
    new Registration { StudentId = 1, CourseId = 101, DateRegistered
        = new DateTime(2018, 2, 15) },
    new Registration { StudentId = 1, CourseId = 102, DateRegistered
        = new DateTime(2018, 2, 15) },
    new Registration { StudentId = 2, CourseId = 102, DateRegistered
        = new DateTime(2018, 2, 17) }
};
```

When using query syntax, the LINQ query will be very similar to a matching SQL query:

```
var query = from student in students
            join registration in registrations
            on student.Id equals registration.StudentId
            select new { student.Id, student.FirstName, student.LastName,
            registration.CourseId, registration.DateRegistered };
```

For an inner join, the **join** clause is used. It consists of two parts:

- The first part before the **on** keyword specifies the second data source using the same syntax as the **from** clause.
- The second part after the **on** keyword specifies the criteria for matching the items in the two data sources.

With the method syntax, an inner join query has only a few similarities with the SQL query:

```
var query = students.Join(registrations,
    student => student.Id,
    registration => registration.StudentId,
    (student, registration) => new { student.Id, student.
```

```
FirstName, student.LastName, registration.CourseId, registration.  
DateRegistered });
```

The **Join** method parameters include the second data source, and the delegates for selecting the matching value in both data sources. A separate call to the **Select** method is not needed here. Instead, the projection delegate is passed as the last argument to the **Join** method.

Multiple inner joins can be combined in a single query. Let's add a third data source for this purpose:

```
private readonly StudentGrade[] grades = new[] {  
    new StudentGrade { StudentId = 1, CourseId = 102, GradingDate =  
        new DateTime(2018, 6, 10), Grade = 9 },  
    new StudentGrade { StudentId = 2, CourseId = 102, GradingDate =  
        new DateTime(2018, 6, 10), Grade = 10 }  
};
```

In **query syntax**, the second join simply follows the first one:

```
var query = from student in students  
    join registration in registrations  
    on student.Id equals registration.StudentId  
    join grade in grades  
    on new { registration.StudentId, registration.CourseId } equals  
    new { grade.StudentId, grade.CourseId }  
    select new { student.Id, student.FirstName, student.LastName,  
        registration.CourseId, registration.DateRegistered, grade.Grade  
};
```

In the second inner join, two properties are used for matching the items. In such cases, an anonymous type must be created.

The same approach must be used with **the method syntax**:

```
var query = students.Join(registrations,  
    student => student.Id,  
    registration => registration.StudentId,  
    (student, registration) => new { student.Id, student.FirstName,
```



```

student.LastName, registration.StudentId, registration.CourseId,
registration.DateRegistered })
.Join(grades,
    joined => new { joined.StudentId, joined.CourseId },
    grade => new { grade.StudentId, grade.CourseId },
    (joined, grade) => new { joined.Id, joined.FirstName, joined.
        LastName, joined.CourseId, joined.DateRegistered, grade.Grade
    });

```

Note: There is an important detail when doing multiple joins with method syntax. Since the projection is part of the Join method, any properties required in the second Join method 'must' be included in the projection result of the first Join call. In the above example, I had to include the StudentId property for this reason.

The left outer join is similar to inner join, with one difference. In addition to the result of the inner join, it also includes items from the first data source which don't have a match in the second data source. They are included once, with null values for data missing from the second data source.

Id	FirstName	LastName
1	John	Doe
2	Jane	Roe
3	Max	Lane

StudentId	CourseId	DateRegistered
1	101	2018-02-15
1	102	2018-02-15
2	102	2018-02-17

Id	FirstName	LastName	CourseId	DateRegistered
1	John	Doe	101	2018-02-15
1	John	Doe	102	2018-02-15
2	Jane	Roe	102	2018-02-17
3	Max	Lane	(null)	(null)

Figure 2: Left outer join

To achieve this in LINQ, a different approach is required. An operation called **group join** must be used instead of a regular join:

```

var query = from student in students
    join registration in registrations
    on student.Id equals registration.StudentId into
    registrationsForStudent
    from registrationForStudent in registrationsForStudent
    select new { student.Id, student.FirstName, student.LastName,

```

```
registrationForStudent.CourseId, registrationForStudent.  
DateRegistered };
```

The above query is still an inner join, although it's using a group join.

In contrast to the regular join, the final projection doesn't access the items from the original second data source. Instead it uses the items from the intermediate result (as declared with the **into** clause) of the join which splits the items from the second data source, into separate instances of `IEnumerable`, each containing only matches for a specific item from the first data source.

The approach used is easier to recognize in **method syntax**:

```
var query = students.GroupJoin(registrations,  
    student => student.Id,  
    registration => registration.StudentId,  
    (Student student, IEnumerable<Registration>  
    registrationsForStudent) => new { student,  
    registrationsForStudent })  
    .SelectMany(studentWithRegistrations =>  
    studentWithRegistrations.registrationsForStudent,  
    (studentWithRegistrations, registration) => new {  
        studentWithRegistrations.student.  
        Id, studentWithRegistrations.student.FirstName,  
        studentWithRegistrations.student.LastName, registration.  
        CourseId, registration.DateRegistered }));
```

In contrast to the **Join** method, the projection delegate in the **GroupJoin** method (which is the last parameter of the method) receives an `IEnumerable` of registrations from the second data source, instead of a single registration. This collection is then flattened using the **SelectMany** method call:

- Its first parameter is used to project the student with the registrations (created in the **GroupJoin** projection) into a standalone collection of registrations for each student.
- The second parameter does the final projection. For each student, it is called as many times as there are registrations for her/him in the nested collection projected using the first parameter of the **SelectMany** method. The first parameter of this projection is the student (with all his registrations) from the

collection that the `SelectMany` method operates on (the result of the `GroupJoin` method). The second one is a registration from the nested collection of registrations for the student.

The result of the group join still includes students from the first data source which don't have a matching registration in the second data source. They're only omitted during flattening. To avoid that, the `DefaultIfEmpty` extension method can be used before flattening which will replace an empty collection with a collection having a single null value item:

```
var query = from student in students
            join registration in registrations
            on student.Id equals registration.StudentId into
            registrationsForStudent
            from registrationForStudent in registrationsForStudent.
            DefaultIfEmpty()
            select new { student.Id, student.FirstName, student.LastName,
            registrationForStudent?.CourseId, registrationForStudent?.
            DateRegistered };
```

In this example, students with no matching registrations will have a null value in the registration variable. That's why I'm using the null conditional operator in the projection delegate.

With the method syntax, the exact same approach can be used:

```
var query = students.GroupJoin(registrations,
    student => student.Id,
    registration => registration.StudentId,
    (Student student, IEnumerable<Registration>
    registrationsForStudent) => new { student,
    registrationsForStudent })
    .SelectMany(studentWithRegistrations =>
    studentWithRegistrations.registrationsForStudent.
    DefaultIfEmpty(),
    (studentWithRegistrations, registration) => new {
    studentWithRegistrations.student.Id,
    studentWithRegistrations.student.FirstName,
    studentWithRegistrations.student.LastName,
    registration?.CourseId, registration?.DateRegistered });
```

The left outer join syntax in LINQ is quite complex and can be confusing if you only take a quick look at it. However, once you understand what is happening during processing, you won't find it that difficult to understand and memorize anymore.

52

Q52. Which Set manipulation operations are available in LINQ?

There is a full offering of LINQ extension methods for Set operations. They're not limited to classes implementing the `ISet<T>` interface. Like all the LINQ methods, they can be used with any class implementing the `IEnumerable<T>` interface.

All set operations return an `IEnumerable<T>`.

The result of the **Union** method is a sequence of items which appear in any of the input collections. If they appear in both collections, they're included only once in the result:

```
var list1 = new[] { 1, 2, 3 };
var list2 = new[] { 3, 4, 5 };

var query = list1.Union(list2); // = { 1, 2, 3, 4, 5 }
```

The **Intersect** method on the other hand returns only the items which appear in both input collections. Again, they're only included once in the result:

```
var list1 = new[] { 1, 2, 3 };
```

```
var list2 = new[] { 3, 4, 5 };

var query = list1.Intersect(list2); // = { 3 }
```

Unlike the previous two methods, the **Except** method is not symmetric. It takes the first collection as the basis, but skips any items in it which also appear in the second collection:

```
var list1 = new[] { 1, 2, 3 };
var list2 = new[] { 3, 4, 5 };

var query = list1.Except(list2); // = { 1, 2 }
```

The last method returning an `IEnumerable<T>` operates only on a single input collection. The **Distinct** method removes all duplicates. The result includes each item only once, even if it appears multiple times in the input collection:

```
var list = new[] { 1, 2, 1, 3, 1, 4, 1, 5 };

var query = list.Distinct(); // = { 1, 2, 3, 4, 5 }
```

All of the methods listed so far have overloads with an additional comparer parameter. If specified, it is used to determine if two items should be considered equal. This can be useful when you can't or don't want to override the **Equals** method of the class in the collection. For example, you might want to keep the default Equals method implementation (which only treats the same instance as equal) but be able to compare the instances based on the value of one or more properties.

The comparer must implement the `IEqualityComparer<T>` interface:

```
public class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Id == y.Id;
    }
}
```

```

public int GetHashCode(Person obj)
{
    return obj.Id.GetHashCode();
}
}

```

Just like when overriding the Equals method, the GetHashCode method must match the logic in the Equals method, i.e. it must return the same value for instances which are considered equal by the Equals method.

An instance of such a comparer can be passed to the Distinct method, for example:

```

var persons = new[]
{
    new Person { Id = 1 },
    new Person { Id = 2 },
    new Person { Id = 1 }
};

var query = persons.Distinct(new PersonComparer()); // .Count() ==
2

```

Because of the comparer, the two instances having the same value of the **Id** property were considered equal, and only one of them was included in the result. If I don't pass in my comparer, all three items will be included in the result (assuming the Equals method is not overridden):

```

var query = persons.Distinct(); // .Count() == 3

```

Set operations available as LINQ extension methods aren't a full replacement for specialized collections implementing the `ISet<T>` interface. They can be very useful when a different type of collection (not implementing `ISet<T>`) is required most of the time, but occasionally a single set operation must be performed.

However, LINQ extension methods don't modify the original collection. Their result can only be persisted in a new collection if necessary. In contrast, set operations in classes implementing the `ISet<T>` interface modify the existing collection. And even when items are added to that collection using the **Add** method, the **Set** semantics are preserved, i.e. no item can appear more than once in the collection.

53

Q53. How can LINQ query results be aggregated?

All aggregation methods process a collection of items and return a single value. LINQ provides a selection of specialized aggregation methods and a very flexible generically named **Aggregate** method which you can use to implement your own custom aggregation logic.

The simplest aggregation method is **Count**. It will return the number of items in the given collection:

```
var list = new[] { 1, 2, 3, 4, 5 };  
  
var count = list.Count(); // = 5
```

There's an overload available accepting a predicate delegate. In this case, only items matching the predicate will be counted:

```
var count = list.Count(x => x % 2 == 0); // = 2
```

The Count method returns an **int** value. If the result exceeds the maximum supported value of that data type ($2^{31} - 1 = 2,147,483,647$), an `OverflowException` is thrown. To

avoid that, the `LongCount` method can be used instead, which returns a `long` value.

The other built-in specialized aggregation methods perform the common numeric aggregation operations. They return the minimum value, the maximum value, calculate the sum or calculate average of all the items in the collection:

```
var list = new[] { 1, 2, 3, 4, 5 };

var min = list.Min(); // = 1
var max = list.Max(); // = 5
var sum = list.Sum(); // = 15
var average = list.Average(); // = 3
```

There are overloads defined for all numeric types. For non-numeric types, the overload with a transform delegate parameter must be used. The delegate extracts or calculates the numeric value from the items to be aggregated:

```
var persons = new[]
{
    new Person { FirstName = "John", LastName = "Doe", Age = 33 },
    new Person { FirstName = "Jane", LastName = "Roe", Age = 24 },
    new Person { FirstName = "Max", LastName = "Lane", Age = 42 }
};

var minAge = persons.Min(person => person.Age); // = 24
```

If none of the methods described so far solves your aggregation requirements, you'll need to use the `Aggregate` method. Its most important parameter is the function used to aggregate the items by combining the aggregated value so far with the current item:

```
static T Aggregate<T>(this IEnumerable<T> source, Func<T, T, T>
func);
```

There are method overloads available with two auxiliary parameters:

- The initial aggregated value (also called seed).
- A projection delegate which converts the final aggregated value into the result.

The following code uses all the parameters:

```
var persons = new[]
{
    new Person { FirstName = "John", LastName = "Doe", Age = 33 },
    new Person { FirstName = "Jane", LastName = "Roe", Age = 24 },
    new Person { FirstName = "Max", LastName = "Lane", Age = 42 }
};

var youngestFullName = persons.Aggregate<Person, Person,
string>(seed: null,
    (youngest, person) => youngest == null || person.Age < youngest.
    Age ? person : youngest,
    youngest => youngest != null ?
    $"{youngest.FirstName} {youngest.LastName}" :
    null); // = "Jane Roe"
```

The method call above returns the full name of the youngest person in the collection. It initializes the result to `null`, indicating that no youngest person has been found yet. The second parameter contains the core logic:

- It returns the current person if it's the first one in the collection or if it is younger than the youngest one so far.
- Otherwise, it returns the already found youngest person so far, i.e. if the current person is of the same age or older.

The final parameter concatenates the first and last name of the found person into its full name, correctly handling the null value which would be the final aggregated value for an empty collection.

54

Q54. How can LINQ query results be grouped?

An important part of SQL queries is the ability to group the query results by one or more columns.

LINQ supports this, both in query and in method syntax. As a bonus, the results need not be flattened before they are returned. They can keep their two-level structure created with grouping.

In the **query syntax**, the **group by** clause is used to group the results. The resulting query is very similar to a SQL query:

```
var query = from word in words
            group word by word.Length;
```

In the **method syntax**, the **GroupBy** method is used instead:

```
var query = words.GroupBy(word => word.Length);
```

In both cases, the result is typed as `IEnumerable<IGrouping<int, string>>`. The `IGrouping<TKey, TSource>` might be unfamiliar, so let's look at how we can iterate over

the results:

```
foreach (var group in query)
{
    Console.WriteLine($"{group.Key}:");
    foreach (var item in group)
    {
        Console.WriteLine($"* {item}");
    }
}
```

The outer loop just iterates over the `IEnumerable<IGrouping<TKey,TSource>>`. For each grouping of type `IGrouping<TKey,TSource>`, we access:

- The value which was used for grouping through the `Key` property.
- The items in the group by iterating through the `IEnumerable<TSource>` interface implemented by `IGrouping<TKey,TSource>`.

Let's run the queries on the following input:

```
var words = new[] { "chair", "bench", "table", "lamp" };
```

The above code outputs the following result:

```
5:
* chair
* bench
* table
4:
* lamp
```

In SQL queries, the `GROUP BY` keyword is often followed by `HAVING` keyword to further filter the created groups. This can also be achieved in LINQ queries with the `where` clause:

```
var query = from word in words
            group word by word.Length into grouping
            where grouping.Key >= 5
            select grouping;
```

To get access to the groups inside the query, the group by clause must be extended with the **into** part. It assigns a name to each generated group which can then be used in the **where** clause. In the query syntax, a query may not end with a **where** clause. A **select** clause must follow and specify the return value. In the previous query which ended with the **group by** clause, the same return value was specified implicitly.

The query we just saw will filter out the groups of words with less than 5 letters and only keep the groups of words with 5 or more letters. Using the same code as before to iterate through the result, we would get the following output:

```
5:
* chair
* bench
* table
```

The same query in the method syntax is even simpler:

```
var query = words
    .GroupBy(word => word.Length)
    .Where(group => group.Key >= 5);
```

When using the method syntax, there's an alternative to using the **GroupBy** method for grouping. The **ToLookup** method can be used instead (there's no equivalent for the query syntax):

```
var lookup = words
    .ToLookup(word => word.Length);
```

The result will still implement `IEnumerable<IGrouping<int, string>>`. In addition to that it will also implement `ILookup<int, string>`. This is an interface very similar to `IDictionary<int, IEnumerable<string>>`. Instead of only iterating over the results, other operations are also supported:

- Checking whether a key is present in the result:

```
Console.WriteLine(lookup.Contains(4)); // true
```

- Efficiently accessing a group by its key value:

```
foreach (var item in lookup[5])
```

```
{  
    Console.WriteLine($"{item}");  
}
```

While this could also be achieved using the LINQ extension methods on the result of the `GroupBy` method, there's an important distinction between the two approaches:

- When using the `ToLookup` method, the result is persisted in an instance of the `Lookup<TKey, TSource>` class which implements the `ILookup<TKey, TSource>` interface.
- When using the `ToGroup` method, the result is recalculated every time it is iterated through, either directly or via any of the LINQ extension methods.

Of course, the first option is more performant, especially for large source collections.

55

Q55. How can LINQ query results be reused without reevaluating?

By default, LINQ queries are not executed immediately at the point of declaration. They only describe how the result can be calculated, but the actual evaluation is postponed until the values from the result are accessed, e.g. by iterating through the result:

```
var list = new List<int> { 1, 2, 3 };
var query = list.Select(item => item * 2);
list.Add(4);

foreach (var item in query)
{
    Console.WriteLine($"{item} "); // 2 4 6 8
}
```

In the above example, the source collection is modified after the query has been declared but before iterating through the result. Because the evaluation happens only when iterating through the result, the output includes the item which was added after the query was defined.

If the query is iterated through multiple times, it will be evaluated again every time:

```
var list = new List<int> { 1, 2, 3 };
var query = list.Select(item => item * 2);

foreach (var item in query)
{
    Console.WriteLine($"{item} "); // 2 4 6
}

list.Add(4);

foreach (var item in query)
{
    Console.WriteLine($"{item} "); // 2 4 6 8
}
```

The two loops through the result in the above example give different outputs because the data source has been modified between the first and the second loop. The value corresponding to the newly added item is included in the second output.

Such behavior is often desired.

The deferred execution postpones all the processing as much as possible. The calculations or I/O operations involved in evaluating an item are only performed when that item in the result set is accessed. This distributes the performance impact of the evaluation over the full duration of enumerating the items in the result set (instead of doing all the processing in one batch, evaluating all the items and storing them in memory, before the first one of them can be accessed).

If for some reason some items from the query never get accessed, they won't be evaluated at all.

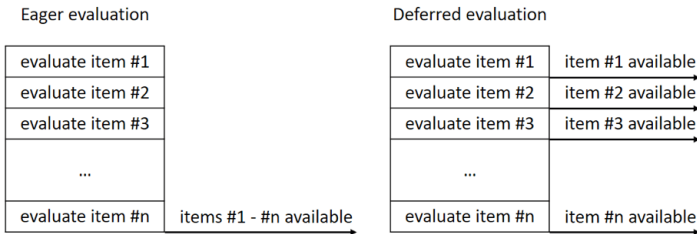


Figure 1: Availability of items in eager and deferred execution models

Still, deferred execution has its disadvantages and sometimes immediate (also called

eager) evaluation is preferred. For such scenarios, LINQ provides the means to execute the query in full, when requested.

The most commonly used LINQ methods to achieve that are **ToList** and **ToArray**. They perform the iteration through the query result to force the evaluation and store the items in a local data structure, List or array respectively, as evident from their names:

```
var list = new List<int> { 1, 2, 3 };
var result = list.Select(item => item * 2).ToList();
list.Add(4);

foreach (var item in result)
{
    Console.WriteLine($"{item} "); // 2 4 6
}
```

In this example, the query declaration is immediately followed by a call to the **ToList** method. Because of that, the query is evaluated at that point and the result is stored in a new **List** instance.

The change to the source list afterwards does not affect the stored result any more. Iterating through the result still outputs the values corresponding to the items which were in the source list when the **ToList** method was called.

In spite of the initial performance hit when the result is calculated and the additional memory requirements for storing the result is determined, this approach still has benefits if the result needs to be iterated through multiple times:

- The processing involved in retrieving or generating the result will only happen *once* therefore the total time required for that will be shorter.
- The result will not change from one iteration to another.

By being aware of the differences between the deferred and the immediate execution of LINQ queries, you can always choose the one which better suits your requirements.

It's also important to be aware that the above-mentioned methods for storing the query result in a local data structure are not the only ones forcing immediate evaluation of queries. The same happens when aggregate functions are used:

```
var list = new List<int> { 1, 2, 3 };  
var result = list.Select(item => item * 2).Sum();  
list.Add(4);  
  
Console.Write(result); // 12
```

When the **Sum** aggregation method is called in the above example, the result is calculated immediately. Therefore, the item added later does not affect the result.

All methods not returning an `IEnumerable<T>` (or `IQueryable<T>`) will manifest such behavior. To calculate their return value, they need to iterate through the items and evaluate themselves.

56

Q56. When and how is a LINQ query executed and how does this affect performance?

The key feature of LINQ is its universal querying API independent of the target data source. However, the way LINQ queries are executed depends on the data source being queried.

When querying local in-memory collections (commonly called **LINQ to Objects**), the LINQ extension methods for the `IEnumerable<T>` interface are used. The implementation of the extension methods somewhat depends on whether the method returns a value typed as `IEnumerable<T>`, or a scalar value.

The methods returning an `IEnumerable<T>` execute in a deferred way, i.e. at the point in code where the result is retrieved, not when the query is defined:

```
var list = new List<int> { 1, 2, 3, 4, 5 };
var query = list.Where(item => item < 3); // not executed yet
It's not too difficult to implement such an extension method
ourselves:
public static IEnumerable<T> Where<T>(this IEnumerable<T> list,
Func<T, bool> predicate)
{
    foreach (var item in list)
```

```
{
    if (predicate(item))
    {
        yield return item;
    }
}
```

For a **Where** method, we only need to iterate through all the items using a **foreach** loop and return those items which satisfy the supplied predicate.

To achieve deferred execution, the method is implemented as an iterator, i.e. the values are returned using the **yield** return statement (you can read more about that in Chapter 44 – “How to implement a method returning an `IEnumerable`?”). Whenever this statement is reached, the control of code execution is transferred back to the caller performing the iteration until the next item from the returned `IEnumerable<T>` is requested.

The methods returning a scalar value are in a way even simpler because they execute immediately:

```
var list = new List<int> { 1, 2, 3, 4, 5 };
var sum = list.Sum(); // executes immediately
Such extension methods don't even need to be iterators:
public static int Sum(this IEnumerable<int> list)
{
    var sum = 0;
    foreach (var item in list)
    {
        sum += item;
    }
    return sum;
}
```

For a **Sum** method, we need to iterate through the items with a **foreach** loop to calculate the total sum returned in the end.

The fact that LINQ to Objects extension methods accept `IEnumerable<T>` as their first argument makes them useful not only for querying collection classes, but in other scenarios as well. One such example is querying of XML documents, also known as

LINQ to XML.

XML documents must still be loaded in memory when they are queried, but they are not modelled with standard collection classes. Instead, a root `XDocument` class is used to represent an XML document. Usually the XML document will be read from a file or parsed from an XML string:

```
var xmlDoc = XDocument.Parse(xmlString);
```

The `Root` property will contain an instance of `XElement`, corresponding to the root element of the XML document. This class provides [many methods for accessing the other elements and attributes](#) in the document. The most commonly used among them are probably:

- `Elements`, returning the direct child elements as an instance of `IEnumerable<XElement>`
- `Descendants`, returning direct and indirect child elements as an instance of `IEnumerable<XElement>`
- `Attributes`, returning the attributes of the element as an instance of `IEnumerable<XAttribute>`

Since they all return `IEnumerable<T>`, they can easily be queried using LINQ. They don't provide strong type checking for element and argument names, though. Strings are used instead:

```
var elements = xmlDoc.Root.Elements()
    .Where(element => int.Parse(element.Attribute("age").Value) >
21);
```

For a certain structure of the XML, the above query would return all person elements with the age attribute above a specific value. Here's an example of such a document:

```
<persons>
  <person name="John" surname="Doe" age="33" />
  <person name="Jane" surname="Doe" age="42" />
  <person name="Johnny" surname="Doe" age="13" />
</persons>
```

To use LINQ to XML, we don't need to concern ourselves with the parsing of the XML documents. Since elements and attributes are exposed as instances of `IEnumerable<T>`, we can use the LINQ extension methods on them in combination with the properties available on the `XElement` and `XAttribute` classes.

The abstraction of underlying data source extends over to **LINQ to Entities** as well. This term represents querying of external databases using [Entity Framework](#) or [Entity Framework Core](#). Although the data is not queried locally in this case, the querying experience is almost identical.

Sample code from here on will be based on Entity Framework Core. To use it with Microsoft SQL Server, the [Microsoft.EntityFrameworkCore.SqlServer](#) NuGet package must be installed in your project.

To specify the mapping between the code and the database tables, corresponding properties must be defined in a class deriving from the `DbContext` class. Entity Framework will take care of the connectivity to the remote database:

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

    // maps to the Persons table in database
    public DbSet<Person> Persons { get; set; }
}
```

Rows in each database table are represented by a simple DTO (data transfer object) matching the table structure:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
    public int Age { get; set; }
}
```

The connection string for the database can be specified in a `DbContextOptions<T>`

instance:

```
var optionsBuilder = new DbContextOptionsBuilder<PersonContext>();
optionsBuilder.UseSqlServer(connectionString);
var options = optionsBuilder.Options;
```

With this set up, a database table can be queried very similar to an XML document or a local collection:

```
using (var context = new PersonContext(options))
{
    var query = context.Persons.Where(person => person.Age > 21);
}
```

Because the `DbSet<T>` collections implement `IQueryable<T>`, not just `IEnumerable<T>`; the extension methods on the former interface are used instead. The `IQueryable<T>` interface makes it possible to implement a **LINQ query provider** which takes care of querying the data remotely. In our case, the provider is part of Entity Framework Core.

When writing queries, most of the times we aren't aware of the difference between the two interfaces because the same set of extension methods is available for both of them. They are just implemented differently.

For the above query, the LINQ query provider for SQL Server would send the following SQL query to the server:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].
[Surname]
FROM [Persons] AS [person]
WHERE [person].[Age] > 21
```

At least for simple cases when we are querying the data, it's not important what SQL query is generated and how. We can safely assume that the data will be retrieved correctly and in an efficient manner. We can even use certain common .NET methods in the query predicates:

```
var query = context.Persons.Where(person => Math.Abs(person.Age) >
21);
```

The LINQ provider will still generate a matching SQL query:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].  
[Surname]  
FROM [Persons] AS [person]  
WHERE ABS([person].[Age]) > 21
```

However, this will only work for functions which have built-in equivalents in the target database and can therefore be implemented in such a way by the LINQ provider. If we start using our own custom methods, they of course won't be converted correctly any more:

```
private static int CustomFunction(int val)  
{  
    return val + 1;  
}  
  
var query = context.Persons.Where(person => CustomFunction(person.  
Age) > 21);
```

In this case, the following SQL query will be sent to the database:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].  
[Surname]  
FROM [Persons] AS [person]
```

Although all the rows from the **Persons** table will be retrieved, the LINQ query will still return the correct result. The filtering will be done locally *after* the data is retrieved. The results are still correct, but the performance will suffer when the number of rows in the table increases. Because of that, we should always make the necessary tests to discover such problems during development.

There are many more types of data sources which can be queried using LINQ. A large collection of third party LINQ providers are published on NuGet for querying different types of relational ([NHibernate](#)) and non-relational ([MongoDB.Driver](#)) databases, REST services ([Linq2Rest](#)), and more.

Depending on how they are implemented, there might be differences in how much of the query is executed remotely and which parts of it are processed locally after the

data is received. Therefore it's important to always read the documentation and test the functionality and performance of the final code.

57

Q57. How are lambda expressions used in LINQ queries?

There are two types of LINQ extension methods:

- Extension methods for the `IEnumerable<T>` interface are used when data sources are queried locally.
- Extension methods for the `IQueryable<T>` interface are used when data sources are queried remotely.

Both can accept lambda expressions as arguments for predicates, selectors and other similar parameters. However, there is a difference in how these parameters are typed in the signatures of the extension methods.

The extension methods for the `IEnumerable<T>` interface accept delegates as their parameters, i.e. the requested parameter type is `Func<,>`, e.g. `Func<T, bool>` for predicates:

```
public static IEnumerable<TSource> Where<TSource>(this  
IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

Of course, lambda expressions can be passed as arguments for such parameters:

```
var query = list.Where(item => item < 3);
```

However, the main benefit for using lambda expressions in this case is convenience. The syntax for lambda expressions is shorter and easier to understand compared to other alternatives which work equally well:

- Anonymous methods:

```
var query = list.Where(delegate(int item)
{
    return item < 3;
}));
```

- Named methods:

```
private bool LessThan3(int val)
{
    return val < 3;
}

var query = list.Where(LessThan3);
```

The difference between the three different examples (lambda expressions, anonymous methods and Named methods) is only in syntax. They will all behave the same when executed.

That is not true for the extension methods for the `IQueryable<T>` interface. Their parameters are typed as expressions, not delegates. Their type in the signature is therefore `Expression<Func<T, bool>>`, e.g. `Expression<Func<T, bool>>` for predicates:

```
public static IQueryable<TSource> Where<TSource>(this
IQueryable<TSource> source, Expression<Func<TSource, bool>>
predicate);
```

The difference lies when invoking such a method. Lambda expressions are the only valid parameters in this case:

```
var query = context.Persons.Where(person => person.Age > 21);
```

If we try to use an anonymous method instead, the code will still compile:

```
var query = context.Persons.Where(delegate(Person person)
{
    return person.Age > 21;
});
```

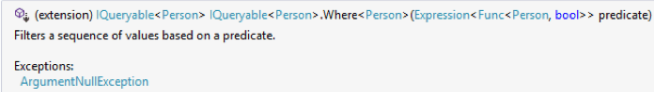
It will also compile with a named method:

```
private bool OlderThan21(Person person)
{
    return person.Age > 21;
}

var query = context.Persons.Where(OlderThan21);
```

However, **the behavior will not be the same**. If we inspect the code in Visual Studio in more detail, we will notice that only in the first example (with the lambda expression argument), the extension method on the `IQueryable<T>` interface is invoked:

```
var query = context.Persons.Where(person => person.Age > 21);
```

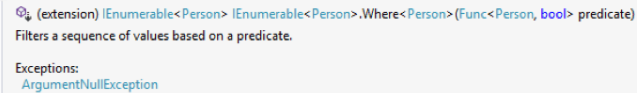


The tooltip shows the signature: `(extension) IQueryable<Person> IQueryable<Person>.Where<Person>(Expression<Func<Person, bool>> predicate)`. It also includes the description "Filters a sequence of values based on a predicate." and the exception `ArgumentNullException`.

Figure 1: `IQueryable<T>.Where` will be invoked

In the latter two examples, the extension method on the `IEnumerable<T>` interface will be invoked instead:

```
var query = context.Persons.Where(delegate(Person person)
{
    return person.Age > 21;
});
```



The tooltip shows the signature: `(extension) IEnumerable<Person> IEnumerable<Person>.Where<Person>(Func<Person, bool> predicate)`. It also includes the description "Filters a sequence of values based on a predicate." and the exception `ArgumentNullException`.

Figure 2: `IEnumerable<T>.Where` will be invoked

This happens because the compiler treats lambda expressions differently from anonymous and named methods. It can convert them into a special data structure describing the expression (`Expression<Func<T, bool>>`, as visible in the Figure 1). They can then be traversed from code using the [expression trees API](#).

The compiler can't do the same with anonymous and named methods. It can only treat

them as delegates. Therefore, it binds the call to the extension method overload with the delegate parameter (as visible in the second image). This overload is defined for the `IEnumerable<T>` interface which the `IQueryable<T>` interface extends.

This difference has an important impact on the behavior though. When the `IQueryable<T>` extension method is invoked, the lambda expression is translated to a remote query:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].[Surname]
FROM [Persons] AS [person]
WHERE [person].[Age] > 21
```

When the `IEnumerable<T>` extension method is invoked, the information from the anonymous and the named method is not included in the remote query:

```
SELECT [p].[Id], [p].[Age], [p].[Name], [p].[Surname]
FROM [Persons] AS [p]
```

Instead, the methods are executed locally after all the data is retrieved from the remote data source. This has important performance implications.

There's a technical reason behind this.

The LINQ query providers need to analyze the `Expression<TDelegate>` data structure describing the lambda expression in order to build the corresponding remote query to execute. Depending on the LINQ provider, this remote query can be a SQL query, a REST call or whatever API the remote data source understands. They cannot do that with a delegate.

To take full advantage of the LINQ query providers, it is therefore important to specify their arguments as lambda expressions, and not as anonymous or named methods.

58

Q58. When should my method return IQueryable<T> and when IEnumerable<T>?

In a larger application, you might want to wrap any direct access to a remote data source in your own methods by implementing the [Repository pattern](#) or by using a similar approach. This will allow you to change the implementation of these methods without affecting the rest of the application.

Such an approach can be very useful in at least two scenarios:

- When testing the business logic, the necessary fake data can be provided by a different repository implementation created specifically for tests.
- A different repository implementation can be created to access a different remote data source or to use a different data access framework to access the same data source.

Since LINQ extension methods for querying data sources with deferred execution return instances of the IQueryable<T> interface, this sounds as an obvious choice for the return value type of the repository method:

```
public static IQueryable<Person>  
GetPersonsQueryable(Expression<Func<Person, bool>> predicate)
```

```
{
    return Context.Persons.Where(predicate);
}
```

Because the `IQueryable<T>` interface extends the `IEnumerable<T>` interface, it would be just as easy to return the latter instead:

```
public static IEnumerable<Person>
GetPersonsEnumerable(Expression<Func<Person, bool>> predicate)
{
    return Context.Persons.Where(predicate);
}
```

Although it might not be obvious, this choice has a few important implications.

It determines to what extent the calling code can affect the generated queries for the remote data source. When an instance of `IQueryable<T>` is returned by the repository method, the caller has almost full control over the generated query by calling additional `IQueryable<T>` extension methods on the returned value:

```
var query = GetPersonsQueryable(person => person.Age > 21)
    .Where(person => person.Name == "Jane");
```

The above call would generate the following SQL query if the SQL Server LINQ provider for Entity Framework was used:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].
[Surname]
FROM [Persons] AS [person]
WHERE ([person].[Age] > 21) AND ([person].[Name] = N'Jane')
```

The generated SQL query does not only incorporate the predicate argument passed to the repository method, but also the predicate from the additional external `Where` method call. If you want to prevent that from happening, you should return an instance of `IEnumerable<T>` instead of the `IQueryable<T>`. The calling code might look the same:

```
var query = GetPersonsEnumerable(person => person.Age > 21)
    .Where(person => person.Name == "Jane");
```

But the generated SQL query will be different:

```
SELECT [person].[Id], [person].[Age], [person].[Name], [person].[Surname]
FROM [Persons] AS [person]
WHERE [person].[Age] > 21
```

Although the same value is returned by the repository method, it is declared to be of a different type. Because of this, the compiler will bind it to the `IEnumerable<T>` extension method instead of the `IQueryable<T>` extension method. As a result, the predicate of the external `Where` method is not accessible to the LINQ provider which therefore cannot include it in the query.

In this specific case, the second example will in most cases perform worse because more data will be retrieved from the database and the outer `Where` method will filter it in-memory afterwards. But the same query could easily be achieved by simply including the condition from the second predicate in the repository method parameter:

```
var query = GetPersonsEnumerable(person => person.Age > 21 &&
person.Name == "Jane");
```

At the first glance it might seem that returning `IEnumerable<T>` instead of `IQueryable<T>` will reduce flexibility. However, the functionalities of any other extension methods which could benefit performance if executed remotely (such as `OrderBy`, `Take`, `Select...`) can also be exposed through the signature of the repository method without having to declare the return value as `IQueryable<T>`:

```
public static IEnumerable<Person>
GetPersonsEnumerable(Expression<Func<Person, bool>> predicate, int
maxResults)
{
    return Context.Persons
        .Where(predicate)
        .Take(maxResults);
}
```

Such approach introduces a higher level of control over what the caller can do and makes the generated queries sent to the data source much more predictable.

You might be concerned that returning an `IQueryable<T>` will make it more difficult to create a repository implementation for testing purposes. Inside a testing repository, we will typically use a local data source which will perform faster and is easier to seed with specific test data. However, when using LINQ with local data sources, the `IEnumerable<T>` interface is used by default and can't be returned directly as an instance of `IQueryable<T>`.

Fortunately, there's a convenient extension method available to convert any `IEnumerable<T>` instance to an `IQueryable<T>` instance which will simply call the corresponding extension methods of the original `IEnumerable<T>` instance:

```
IQueryable<Person> queryableList = new List<Person>().  
AsQueryable();
```

With all of the above in mind, I believe that it's a better idea to return an `IEnumerable<T>` instead of `IQueryable<T>` in most cases. It will make your life easier in the long run.

There are exceptions to this, e.g. internal helper methods for composing queries. Since these methods cannot be accessed directly by external callers, the queries are entirely under your control. Therefore, the above-mentioned advantages of `IEnumerable<T>` don't apply to them.

In the repository methods so far, we have ignored how the `DbContext` instance is instantiated and when it gets disposed. However, because of deferred execution, incorrect handling of its lifetime can quickly cause issues. For example, let's look at the following naïve implementation with the repository method instantiating and disposing the `DbContext`:

```
public static IEnumerable<Person>  
GetPersonsInternalContext(Expression<Func<Person, bool>> predicate)  
{  
    using (var context = new PersonContext())  
    {  
        return context.Persons.Where(predicate);  
    }  
}
```

This wrapped query only gets executed when the result is enumerated, i.e. in the calling code:

```
var result = GetPersonsInternalContext(person => person.Age > 21).
ToList();
```

By then, the `DbContext` instance would have already been disposed, therefore the call will fail with an `ObjectDisposedException` exception.

To avoid this problem, the `using` block must not be exited before the consuming code completes the enumeration of the result. This can be achieved by returning your own `IEnumerable<T>` instance from the method using the `yield return` keywords (you can read more about that in Chapter 44 – "How to implement a method returning an `IEnumerable`?"):

```
public static IEnumerable<Person>
GetPersonsInternalContext(Expression<Func<Person, bool>> predicate)
{
    using (var context = new PersonContext())
    {
        foreach (var item in context.Persons.Where(predicate))
        {
            yield return item;
        }
    }
}
```

An alternative option would be for the caller to take on the responsibility for the `DbContext`. A simple implementation would be to pass in the `DbContext` to the repository method as a parameter:

```
public static IEnumerable<Person> GetPersonsExternalContext(this
PersonContext context, Expression<Func<Person, bool>> predicate)
{
    return context.Persons.Where(predicate);
}
```

Instead of passing the `DbContext` instance to each method call, a more common approach is to pass it as a constructor parameter to the class containing the repository methods. This technique is called [dependency injection](#) and modern frameworks such as [ASP.NET Core](#) already have [built-in support](#) to make it easier to use.

SECTION VII

PARALLEL AND ASYNCHRONOUS PROGRAMMING

59

Q59. What are the differences between Concurrent, Multithreaded and Asynchronous programming?

Concurrent, multithreaded and asynchronous programming are terms that are often mentioned together. What do they mean exactly and how are they related to each other?

Concurrent programming is the broadest term of the three. It means that multiple operations are being executed concurrently, i.e. at the same time. The other two terms mentioned (multithreaded and asynchronous programming) are both subcategories of concurrent programming.

Multithreaded programming means that the program is running in multiple separate threads, concurrently. Individual threads are managed and scheduled to run by the operating system. At any point in time, a CPU core can run a single thread of a particular OS process. If there are more threads to run than there are CPU cores (which is usually the case), the operating system constantly switches between different threads to give the illusion that more of them are running at the same time.

In C#, there are multiple APIs available for multithreaded programming. The most basic low-level one is the **Thread** class which can be used to explicitly create a thread:

```
var thread = new Thread(DoProcess);
thread.Start();
// CPU intensive code
thread.Join();
```

In this example, I create a thread and start it immediately. While that thread is running, I can independently run other code on my existing thread. Once I'm ready, I call the **Join** method which blocks the current thread until the other one completes its work. If the other thread has already completed the processing by then, the execution on the current thread continues immediately. The work for the other thread is specified as a delegate which is passed to the **Thread** class constructor:

```
private void DoProcess()
{
    // CPU intensive code
}
```

Today, direct use of the **Thread** class is discouraged in favor of other available APIs for multithreaded programming. The most recent and recommended abstraction to use is the **Task** class from the Task Parallel Library. Using it, the sample above would be implemented as follows:

```
var task = Task.Run((Action)DoProcess);
// CPU intensive code
task.Wait();
```

The code looks very similar, but the benefit of using the **Task** class over the **Thread** class is the abstraction layer between tasks and threads.

A task doesn't directly represent a thread. The task scheduler at the CLR (Common Language Runtime) level is responsible for scheduling the tasks instead. It manages its own **thread pool** (a group of pre-allocated threads) and schedules the tasks to be executed by the threads in it. This allows it to better optimize the execution of work than the programmer could if she/he directly used the **Thread** class.

Multithreaded programming is typically used for CPU intensive processing, also called CPU-bound work. In contrast, **asynchronous programming** is used for the so-called I/O-bound work. These are file system or network operations which take a long time to complete but don't require any CPU processing during that time. When an I/O

operation is done, a callback is called to notify the caller.

The APIs for asynchronous I/O operations in the .NET framework are based on the Task abstraction which makes the code very similar to the second multithreaded example:

```
var fileInfo = new FileInfo(path);
var buffer = new byte[fileInfo.Length];

using (var stream = new FileStream(path, FileMode.Open, FileAccess.
Read, FileShare.Read, buffer.Length, useAsync: true))
{
    var task = stream.ReadAsync(buffer, 0, buffer.Length);
    // other CPU-bound code
    var bytesRead = task.Result;
}
```

Because the I/O operation is asynchronous, the Task returned by the `ReadAsync` method for most of its duration doesn't run on a thread from the thread pool (as the tasks created with the `Task.Run` method do), nor on any other thread. This allows the CPU to do other work in the meantime instead of idly waiting for the I/O operation to complete.

However, the call to `task.Result` in the sample above will wait for the task to complete unless it has already completed by the time of the call. To avoid that, the `await` keyword can be used. You can learn about that in a forthcoming chapter – "What abstractions for multithreaded programming are provided by the .NET framework?".

60

Q60. Which Timer class should I use and when?

Timers are used when a piece of code needs to be executed on a regular interval or after a certain amount of time. However, there are many different timer classes available in the .NET framework. Each one of them has different features and is useful in different scenarios.

The most basic one is the `Timer` class in the `System.Threading` namespace. It can be configured to invoke a single callback delegate after a specified amount of time. All of its configuration values are provided as constructor parameters:

- the callback delegate to invoke,
- the state to pass into the callback delegate,
- the delay before the first invocation in milliseconds, and
- the interval between subsequent invocations in milliseconds.

```
var timer = new System.Threading.Timer(state =>
{
    // code to execute
}, stateObject, 50, System.Threading.Timeout.Infinite);
```

In the above example, the `Timeout.Infinite` value for the last parameter specifies that the callback will only be called once, not repeatedly. The `state` parameter in the lambda will have the value of the `stateObject` which was passed as the second argument of the constructor.

The callback will be invoked on a thread pool. There's no guarantee that this will be the same thread where it was created initially. If the callback takes longer to execute than the interval duration, it can even run multiple times in parallel on different threads. Because of all that, the delegate code must be written in a thread-safe manner.

The `Change` method can be used to modify the delay and interval at a later time. Calling it will restart the timer. The callback cannot be changed. To stop the timer, either use the `Change` method with `Timeout.Infinite` as its first parameter or the `Dispose` method must be called on it.

The `Timer` class in the `System.Timers` namespace is more flexible. Instead of invoking the callback, it raises an event which can have multiple handlers, but no state can be passed to them. It is mostly configured through its properties:

```
var timer = new System.Timers.Timer(50);
timer.AutoReset = false;
timer.Elapsed += (source, eventArgs) =>
{
    // code to execute
};
timer.Start();
```

The timer is initially disabled. The `Start` method must be called to enable it. It can be stopped using the `Stop` method and restarted by calling the `Start` method again.

Instead of calling these two methods, the `Enabled` property can be set to the appropriate value. The `AutoReset` property specifies whether the `Elapsed` event will be invoked only once or multiple times. Of course, event handlers can be added or removed while the timer is running.

By default, event handlers will be invoked on a thread pool thread, therefore the code must be thread-safe just like in the case of the `System.Threading.Timer` class. However, by setting the `SynchronizingObject` property to an instance of a class implementing the `ISynchronizeInvoke` interface (e.g. a Windows Forms form), the event handler calls can be invoked on a specific thread. Although this feature allows the `System.Timers.Timer` event handlers to safely interact with UI components, there are classes better suited to do that in the .NET framework.

The `Timer` class in the `System.Windows.Forms` namespace is specifically designed for use in Windows Forms applications. The event handlers for its `Tick` event will always be invoked on the UI thread.

When following the Windows Forms coding practices, the `Timer` component should be added to a host form using the designer. Its properties will also be set there. The graphical user interface of the designer hides the code generated to match the configured settings. If you're curious, you can find this code in the file named `<FormName>.Designer.cs`. It will be similar to the following:

```
this.timer = new System.Windows.Forms.Timer(this.components);
this.timer.Enabled = true;
this.timer.Interval = 50;
this.timer.Tick += new System.EventHandler(this.timer_Tick);
```

The event handler code will still need to be written manually in the `<FormName>.cs` file. A double-click on the component in the designer will add an empty event handler to it, similar to the following:

```
private void timer_Tick(object sender, EventArgs e)
{
    // code to execute
}
```

The public interface of this `Timer` class is very similar to the one in the `System.Timers` namespace. However, its event has a different name and signature, and there is no `AutoStart` property. This makes it impossible to configure this timer to be invoked only once. To achieve that, the `Stop` method must be called in the event handler.

The WPF (Windows Presentation Foundation) equivalent of this Windows Forms timer is the `DispatcherTimer` class in the `Windows.System.Threading` namespace. The initialization code will be very similar, but it will need to be hand-written in full

because there's no designer support for this class:

```
var timer = new DispatcherTimer();
timer.IsEnabled = true;
timer.Interval = new TimeSpan(50);
timer.Tick += (source, EventArgs) =>
{
    // code to execute
};
```

The public interface is almost identical to the `System.Windows.Forms.Timer` class. You can notice two differences in the above sample code:

- Instead of `Enabled`, the property is named `IsEnabled`.
- The `Interval` property is of type `TimeSpan` instead of `int`.

Although in theory all the functionalities could be implemented by only using the `System.Threading.Timer` class, other timer classes in the .NET framework can make our lives easier when used in specific scenarios because we can take advantage of their additional features.

61

Q61. How to create a New Thread and manage its Lifetime?

While writing new code today, you should avoid managing threads manually because there are better alternatives available (you can read more about these alternatives in the next chapter 62 – "What abstractions for multithreaded programming are provided by the .NET framework?").

Nevertheless, there's still a lot of legacy code that uses the `Thread` class directly and understanding it can prove useful if you ever need to maintain such code.

.NET applications run on a *single* thread by default.

Additional threads can be spawned to allow processing in parallel. The operating system is responsible for scheduling these threads to be executed by the available CPU cores.

When creating a new thread, a delegate must be passed to the constructor. Two different signatures are valid for the delegate.

A simpler version of the thread delegate accepts no parameters:

```
private void NonParameterized()
{
    // do processing
}
```

In this case, the parameterless overload of the `Thread.Start` method is used to start the thread:

```
var thread = new Thread(NonParameterized);
thread.Start();
```

If any parameters need to be passed to the thread delegate, the overload that takes a single parameter of type `object` can be used:

```
private void Parameterized(object input)
{
    // do processing
}
```

The thread delegate parameter will have the value of the argument passed to the `Thread.Start` method:

```
var thread = new Thread(Parameterized);
thread.Start(input);
```

Since the input parameter type is `object`, a value of any type can be passed. Of course, this type should match what the code in the thread delegate expects, so that it can cast the value correctly before accessing it.

For example, the input parameter can be used to pass a callback into the thread delegate. The callback can be invoked to report the result, for example:

```
int result = 0;
Action<int> callback = output => result = output;
var thread = new Thread(ReturnResult);
thread.Start(callback);
```

In the thread delegate, the callback will be casted correctly and invoked once the result is calculated:

```
private void ReturnResult(object input) {
    var callback = (Action<int>)input;

    // do processing

    callback(result);
}
```

In the most common scenario, the thread will simply complete its processing in full. If the invoking thread needs to wait for that (e.g. to make sure that the result has been returned), it can invoke the **Join** method:

```
thread.Join();
```

This call will block until the thread instance it was invoked on, has completed. A timeout parameter can be passed to the **Join** method which specifies the maximum time to wait for the thread to complete:

```
var joined = thread.Join(timeout);
```

The return value indicates whether the thread has completed in the given time or not.

To prematurely stop the processing in a thread, the **Abort** method can be called on it:

```
thread.Abort();
```

This will cause a `ThreadAbortException` exception to be thrown in the thread delegate. If that exception is unhandled, the thread execution will stop immediately. However, if the exception is handled in the thread delegate, the code in the **catch** block will get executed and can be used to do any clean up, before exiting:

```
try
{
    // do processing
}
catch (ThreadAbortException)
{
    // clean up after abort
}
```

To wait for this clean up to complete, the calling thread should call the `Join` method after calling the `Abort` method:

```
thread.Abort();  
thread.Join();
```

After the catch (or finally) block in the thread delegate completes, the `ThreadAbortException` exception will be rethrown, preventing any code after that block to be executed.

To prevent that, the `Thread.ResetAbort` method can be called in the catch or finally block.

However, this should only be done when you also own the code that created the thread. Otherwise, the caller won't expect the execution of the thread to continue after the `Abort` method was called. A following call to the `Join` method might therefore block for an unexpectedly long time.

62

Q62. What abstractions for Multithreaded programming are provided by the .NET framework?

.NET Framework 4 introduced the [Task Parallel Library \(TPL\)](#) as the preferred set of APIs for writing multithreaded code.

To run a piece of code in the background, you need to wrap it into a task:

```
var backgroundTask = Task.Run(() => DoComplexCalculation(42));  
// do other work  
var result = backgroundTask.Result;
```

The `Task.Run` method accepts a `Func<T>` if that code needs to return a result, or an `Action` if it doesn't return any result.

Of course, in both cases you can use a lambda, just as I did in the example above. This allowed me to invoke the long running method with a parameter.

A thread from the thread pool will process the task. The CLR (Common Language Runtime) includes a default scheduler that takes care of queuing and executing the

tasks using threads from the thread pool. Instead of the default scheduler, you can implement your own scheduling algorithm by deriving from the `TaskScheduler` class and using it. This topic is beyond the scope of this book.

Having the task scheduled to run on a thread from the thread pool takes away direct control over the executing thread. The task scheduler's heuristics will in most cases achieve better performance than manual management of threads would. The scheduler is designed to adapt to the current executing environment and tailor the number of threads in the pool to the available resources.

In the sample we just saw, I accessed the `Result` property to use the result of the code running in the background thread, in the calling thread. For tasks that do not return a result, I could instead call the `Wait` method to wait for the background code to complete. Both approaches will block the calling method until the background task completes.

For local processing of collections, the LINQ API provides a simple abstraction for distributing the work over multiple threads. It is called Parallel LINQ (PLINQ). Using it is as simple as just calling the `AsParallel` method on the data source (which implements the `IEnumerable` interface):

```
var query = list.AsParallel()
    .Select(item => DoComplexCalculation(item));
```

The above code will produce the same result as the version without the `AsParallel` method call:

```
var query = list.Select(item => DoComplexCalculation(item));
```

However, instead of processing the items sequentially, it will partition the data in the background and distribute the load over multiple threads running on separate CPU cores. Mostly, no additional changes to the original LINQ code are required because the `AsParallel` method returns an instance of `ParallelQuery<T>` for which the same LINQ extension methods are available, as available for the `IEnumerable<T>` and `IQueryable<T>` interfaces.

The effect of PLINQ will depend on the workload. The more computationally expensive the processing is, the larger the benefit of running it in parallel. Of course, performance also depends on the number of CPU cores. By default, PLINQ tries to use all the cores, but this can be configured programmatically:


```
var query = list.AsParallel().WithDegreeOfParallelism(2)
    .Select(item => DoComplexCalculation(item));
```

The PLINQ implementation assumes that calculations for individual items are independent. If that's not true, the result of the query might not be correct. This of course also means that no shared resources should be accessed in these calculations. If they are, they must be protected using standard locking mechanisms (e.g. the **lock** statement) which will probably negate most of the benefits of using PLINQ.

Even when the processing code adheres to these assumptions, the result might be different because of the parallel processing with PLINQ. By default, PLINQ will not preserve the order of the items processed. This not only means that the order of items will be different in the result, but also that in certain cases, the result itself could be different, e.g. when using the **Take** method to limit the number of items in the result:

```
var query = list.AsParallel()
    .Select(item => DoComplexCalculation(item))
    .Take(10);
```

With sequential LINQ, the order of items in the source list would be preserved. The result would consist of the first 10 items in the source list processed by the **DoComplexCalculation** method. Because PLINQ doesn't preserve the order, there's no guarantee that the above query will return the same 10 items. It will return the 10 items from the source list which will be processed first, in the order they were processed.

When the order of items is important, this can be communicated to PLINQ with the **AsOrdered** method:

```
var query = list.AsParallel().AsOrdered()
    .Select(item => DoComplexCalculation(item))
    .Take(10);
```

The query will now return the same result as sequential processing with LINQ would. It will have a negative impact on the performance though. Therefore, the **AsOrdered** method should only be used when order is important.

63

Q63. What is the recommended Asynchronous Pattern in .NET?

Since version 4, the recommended pattern for asynchronous programming in the .NET framework is **task-based asynchronous programming (TAP)**.

As the name implies, it is based on the **Task** class introduced with the [Task Parallel Library \(TPL\)](#). A task represents an operation running in the background (either asynchronously or on a different thread):

```
var backgroundTask = Task.Run(() => DoComplexCalculation(42));
```

By invoking the **Wait** method or accessing the **Result** property on the **Task** class, the invoking thread can wait for a task:

```
var result = backgroundTask.Result;
```

However, this will block the invoking thread in the meantime.

Since C# 5 (and .NET framework 4.5), the **await** keyword is a better alternative:

```
var result = await backgroundTask;
```

The execution of the code following the `await` call will still continue only after the awaited task completes, but the thread won't idly wait until that happens. Instead, it will be released to do other work. Once the awaited task completes, the code after it will continue to be executed.

The TPL provides helper methods for orchestrating tasks. The simplest one of them is the `ContinueWith` method. When used, asynchronous execution can continue with another method passed in as a delegate:

```
var compositeTask = Task.Run(() => DoComplexCalculation(42))
    .ContinueWith(previous => DoAnotherComplexCalculation(previous.
Result));
```

The same can be achieved in a more readable way, using the `await` keyword:

```
var intermediateResult = await Task.Run(() =>
DoComplexCalculation(42));
var result = DoAnotherComplexCalculation(intermediateResult);
```

There is an important difference in behavior, though.

The line of code after the awaited task will be executed *only* if that task completed successfully. The delegate of the `ContinueWith` method will be executed even if the task ends prematurely as canceled or faulted because of an exception. Some overloads of the `ContinueWith` method accept an additional parameter specifying under which conditions the continuation should run:

```
var initialTask = Task.Run(() => DoComplexCalculation(42));

var successfulContinuation = initialTask.ContinueWith(previous =>
    DoAnotherComplexCalculation(previous.Result),
    TaskContinuationOptions.OnlyOnRanToCompletion);
var failedContinuation = initialTask.ContinueWith(previous =>
    HandleError(previous.Exception), TaskContinuationOptions.
OnlyOnFaulted);
var canceledContinuation = initialTask.ContinueWith(previous =>
    HandleCancelation(), TaskContinuationOptions.OnlyOnCanceled);
```

Since C# 6, the standard error handling mechanisms in the language can be used to achieve the same behavior:

```
try
{
    var intermediateResult = await Task.Run(() =>
DoComplexCalculation(42));
    var result = DoAnotherComplexCalculation(intermediateResult);
}
catch (TaskCanceledException)
{
    await HandleCancelation();
}
catch (Exception exception)
{
    await HandleError(exception);
}
```

Since this code is much easier to read and understand, the `ContinueWith` method is rarely used today.

If you need to run more than one task concurrently, there are methods available to help you coordinate them. To use these methods, you should start all the tasks immediately and collect references to them, e.g. in an array:

```
var backgroundTasks = new[]
{
    Task.Run(() => DoComplexCalculation(1)),
    Task.Run(() => DoComplexCalculation(2)),
    Task.Run(() => DoComplexCalculation(3))
};
```

The static helper methods will allow you to wait for all the tasks to complete, either synchronously or asynchronously:

```
// wait synchronously
Task.WaitAll(backgroundTasks);
// wait asynchronously
await Task.WhenAll(backgroundTasks);
```

If you only need one of the methods to complete before continuing, you have a different pair of methods at your disposal:

```
// wait synchronously
Task.WaitAny(backgroundTasks);
// wait synchronously
await Task.WhenAny(backgroundTasks);
```

Since tasks can be long running, you might want to be able to cancel them prematurely. To allow this option, pass a cancellation token to the method creating the task. You can use it afterwards to trigger the cancellation:

```
var tokenSource = new CancellationTokenSource();
var cancellableTask = Task.Run(() => {
    for (int i = 0; i < 100; i++)
    {
        if (tokenSource.Token.IsCancellationRequested)
        {
            // clean up before exiting
            tokenSource.Token.ThrowIfCancellationRequested();
        }
        // do long-running processing
    }
    return 42;
}, tokenSource.Token);
// cancel the task
tokenSource.Cancel();
try {
    await cancellableTask;
}
catch (OperationCanceledException) {
    // handle cancellation
}
```

If you are the one implementing the task, you need to add additional code to support cancellation. To actually stop the task early, you need to check the cancellation token in the task and react if a cancellation was requested: do any clean up you might need to do and then call the `ThrowIfCancellationRequested` method to exit the task. This will throw an `OperationCanceledException` exception, which can then be handled

accordingly in the calling thread.

It might seem redundant to also pass the cancellation token as an argument to the **Task.Run** method. However, this serves two purposes:

1. If the cancellation has been requested before **Task.Run** is called, then the task will not run at all. Instead it will be immediately put in the Canceled state.
2. If the cancellation is requested while the task is running, it will ensure that the task will transition to the Canceled state instead of the Faulted state when the **OperationCanceledException** is thrown by the executing code.

To take full advantage of the task-based asynchronous pattern, it's important that the APIs in the .NET framework implement this pattern themselves. Since .NET framework 4, asynchronous methods returning tasks have been added to existing classes for I/O operations. They return a **Task** and can usually be easily identified by the **Async** postfix in their names.

For example, the **FileStream** class provides methods for performing operations on a stream asynchronously:

```
using (FileStream srcStream = new FileStream(srcFile, FileMode.  
Open),  
    destStream = new FileStream(destFile, FileMode.Create))  
{  
    await srcStream.CopyToAsync(destStream);  
}
```

In some other cases, new classes have been introduced with such methods. The **HttpClient** class is a newer alternative to the old **WebClient** class. It provides asynchronous methods for HTTP network operations:

```
using (var httpClient = new HttpClient())  
{  
    var content = await httpClient.GetStringAsync(url);  
}
```

When writing fresh code, the task-based asynchronous pattern should be used whenever possible. The only exception to this should be calls to older asynchronous APIs which don't have alternatives following this pattern.

64

Q64. What is the Asynchronous Programming Model?

The asynchronous programming model (APM) was the first asynchronous pattern in the .NET framework.

Since the task-based asynchronous pattern (TAP) is a full modern replacement for it, you won't be implementing your own methods following this pattern anymore. You'll only need to consume existing methods implemented with this pattern for which there are no more modern alternatives available.

The pattern can easily be recognized by the naming convention of the methods. They are always in pairs named `Begin<Postfix>` and `End<Postfix>` where `<Postfix>` matches the name of the synchronous method for the same operation (e.g. the `BeginRead` and `EndRead` methods are asynchronous versions of the `Read` method). The simplest way to use these methods is by just calling them consecutively:

```
var fileInfo = new FileInfo(filename);  
var buffer = new byte[fileInfo.Length];  
using (var stream = new FileStream(filename, FileMode.Open))  
{
```

```
var asyncResult = stream.BeginRead(buffer, 0, buffer.Length,
null, null);
// execute other code in the meantime
var bytesRead = stream.EndRead(asyncResult); // blocks until
completion
}
// use buffer contents
```

Between the two calls, you can add any other code which does not need the result of the operation, yet. This code will be executed in parallel with the asynchronous operation.

However, the call to the **End** method will block the thread until the operation is completed. The call will return the same result as the equivalent synchronous method would. It will also throw an exception in the same manner if an error happens.

To avoid calling the **End** method before the execution completes, you can wait for the handle returned by using the **Begin** method in the `AsyncWaitHandle` property of the `IAsyncResult` instance:

```
var fileInfo = new FileInfo(filename);
var buffer = new byte[fileInfo.Length];
using (var stream = new FileStream(filename, FileMode.Open))
{
    var asyncResult = stream.BeginRead(buffer, 0, buffer.Length,
    null, null);
    // execute other code in the meantime
    if (asyncResult.AsyncWaitHandle.WaitOne(timeout)) // blocks
    temporarily
    {
        asyncResult.AsyncWaitHandle.Close();
        var bytesRead = stream.EndRead(asyncResult); // executes
        immediately
    }
    else
    {
        // the operation has not completed until timeout
    }
}
// use buffer contents
```


Although the call to the `End` method now won't be blocking anymore because the operation would have already completed when the method is called, the behavior won't be much different. The call to the `WaitOne` method will still block the thread.

In certain scenarios, it might nevertheless be beneficial to use the handle instead of directly invoking the `End` method when you need to wait for the asynchronous operation to complete.

For example, you can specify a timeout value which restricts the maximum wait time for the operation to complete. This will prevent blocking the thread for an unexpectedly long duration, and allow you to do something else if the operation doesn't complete in the expected time.

If you are invoking more than one asynchronous operation in parallel, you can also use the `WaitAll` and `WaitAny` static methods on the `WaitHandle` class passing them multiple handles to orchestrate the different asynchronous operations.

Instead of temporarily blocking, you can also periodically poll whether the operation has already completed and do something else until it does complete. The polling itself can be implemented without blocking:

```
var fileInfo = new FileInfo(filename);
var buffer = new byte[fileInfo.Length];
using (var stream = new FileStream(filename, FileMode.Open))
{
    var asyncResult = stream.BeginRead(buffer, 0, buffer.Length,
    null, null);
    while (!asyncResult.IsCompleted)
    {
        // execute other code in the meantime
    }
    var bytesRead = stream.EndRead(asyncResult); // executes
    immediately
}
// use buffer contents
```

If you don't need to access the result of the asynchronous operation synchronously in the invoking code, you can also use a callback:

```
var fileInfo = new FileInfo(filename);
var buffer = new byte[fileInfo.Length];
var stream = new FileStream(filename, FileMode.Open);
stream.BeginRead(buffer, 0, buffer.Length, asyncResult =>
{
    var bytesRead = stream.EndRead(asyncResult); // executes
immediately
    stream.Dispose();
    // use buffer contents
}, null);
// immediately continue executing other code
```

In this case, the callback will be called when the operation completes. The operation result must be processed in the callback, as well as the cleanup after the operation (disposing the stream in our case).

However, the most convenient way for invoking operations which follow the asynchronous programming model is to convert them into tasks and consume them using the task-based asynchronous pattern. There's a helper method available for that:

```
var fileInfo = new FileInfo(filename);
var buffer = new byte[fileInfo.Length];
using (var stream = new FileStream(filename, FileMode.Open))
{
    var bytesRead = await Task<int>.Factory.FromAsync(stream.
BeginRead,
    stream.EndRead, buffer, 0, buffer.Length, null); // releases
thread for other work
}
// use buffer contents
```

The `FromAsync` method in the example returns a task which can be awaited. This will release the invoking thread for other work until the asynchronous operation completes, as we're used to. The resulting code will prevent any blocking of the calling thread. At the same time, this is much easier to understand and implement correctly when compared to any of the previous examples.

65

Q65. What is the Event-based Asynchronous Pattern?

The event-based asynchronous pattern (EAP) was the second asynchronous pattern to be introduced to the .NET framework.

Although it is more recent than the asynchronous programming model (APM), it's still considered legacy and there's no reason to implement your own code based on this pattern, instead of the newest task-based asynchronous pattern (TAP). You might still need to consume methods implemented with this pattern if you need APIs which are not (yet) exposed as asynchronous methods returning tasks.

As the name suggests, this pattern features heavy use of events.

The asynchronous method will typically have the *Async* postfix and won't return any value. Instead, there will be a matching event on the class, having the same base part of the name and *Completed* as a postfix (e.g. for a method named `DownloadStringAsync`, there will be an event named `DownloadStringCompleted`).

When the asynchronous method completes, it will raise that event, therefore an event handler should always be added to it:

```
var webClient = new WebClient();
webClient.DownloadStringCompleted += (sender, args) =>
{
    webClient.Dispose();
    var contents = args.Result;
    // process the result
};
webClient.DownloadStringAsync(url);
```

The event handler will retrieve the result from the **Result** property of the `DownloadStringCompletedEventArgs` argument and process it. It will also be responsible for cleanup (e.g. disposing the parent instance if it's not needed anymore).

The asynchronous method won't throw any exceptions. Instead, that exception will be stored in the **Error** property of the event handler's `DownloadStringCompletedEventArgs` argument. This property should always be tested before accessing the result:

```
webClient.DownloadStringCompleted += (sender, args) =>
{
    webClient.Dispose();
    if (args.Error != null)
    {
        // handle error
    }
    else
    {
        var contents = args.Result;
        // process the result
    }
};
```

An asynchronous operation that's still in progress can be cancelled by invoking the **CancelAsync** method on the same instance. In this case, the **Cancelled** property of the event handler's `DownloadStringCompletedEventArgs` argument will be set to **true**. Also, there will be an exception assigned to the **Error** property. Therefore, you should always test the **Cancelled** property first to distinguish between cancelled and faulted operations:

```

webClient.DownloadStringCompleted += (sender, args) => {
    webClient.Dispose();
    if (args.Cancelled)
    {
        // handle cancellation
    }
    else if (args.Error != null)
    {
        // handle error
    }
    else
    {
        var contents = args.Result;
        // process the result
    }
};

```

To avoid having multiple different asynchronous patterns in your code, it's best to convert any calls to methods following the event-based asynchronous patterns into tasks, and then use them according to the task-based asynchronous pattern. The `TaskCompletionSource` class can be used to achieve that:

```

var taskSource = new TaskCompletionSource<string>();
var webClient = new WebClient();
webClient.DownloadStringCompleted += (sender, args) => {
    webClient.Dispose();
    if (args.Cancelled)
    {
        taskSource.SetCanceled();
    }
    else if (args.Error != null)
    {
        taskSource.SetException(args.Error);
    }
    else
    {
        taskSource.SetResult(args.Result);
    }
};

```

```
webClient.DownloadStringAsync(url);  
  
var contents = await taskSource.Task;
```

The key part of the conversion is invoking one of the three methods on the `TaskCompletionSource` instance based on the outcome of the asynchronous operation:

- If the asynchronous method completed successfully, the `SetResult` method should be called with the result of the operation as the argument.
- If an error occurred, the `SetException` method should be called with the exception thrown as the argument.
- If the asynchronous method was cancelled, the `SetCanceled` method should be called.

As you can see, the code structure inside the event handler is almost identical to the one in the previous example. The only difference is that instead of directly handling the cancellation, or handling the error or processing the result, everything is just passed on to the `TaskCompletionSource` instance using the corresponding method calls.

The resulting task can be accessed as the `Task` property on the `TaskCompletionSource` instance. It will behave just like any other task representing an asynchronous operation. When awaited, it will release the thread for other work. The code after the `await` call will only continue executing after the asynchronous operation is completed.

66

Q66. In what order is code executed when using `async` and `await`?

With synchronous (single-threaded) code, the code always executes sequentially, line by line (or statement by statement).

For example, let's look at the following method from a WPF (Windows Presentation Foundation) application:

```
private void OnClickSync(object sender, RoutedEventArgs e)
{
    StatusText.Text = "Downloading...";

    var request = WebRequest.Create(requestedUri);
    using (WebResponse response = request.GetResponse())
    {
        StatusText.Text = string.Empty;

        // process the response
    }
}
```

The following actions will be executed when the method is invoked:

- A new status text will be set, informing the user that an action will take a long time.
- A web request will be created, downloading the contents from the specified URL.
- The status text will be cleared.
- The downloaded contents will be processed further.

Despite the seemingly self-explanatory code, the status text will *never* be shown to the user. To understand why this happens, we must take a closer look at how the contents of the window are rendered in a WPF application.

Instead of applying the changes immediately, they are batched and rendered *at the same time*. Although this makes the rendering more efficient, in our case, delays it until our synchronous method completes the processing. By that time, there is no status text to render (because it was set to an empty string before the method completed).

This happens because each application with a graphical user interface (GUI) has a main UI thread which is responsible for all interaction with the user interface. This includes both processing of the incoming events (e.g. the user clicking a button), as well as rendering. So, until our synchronous method completed, the UI thread was busy with it and couldn't re-render the UI or process any other events:

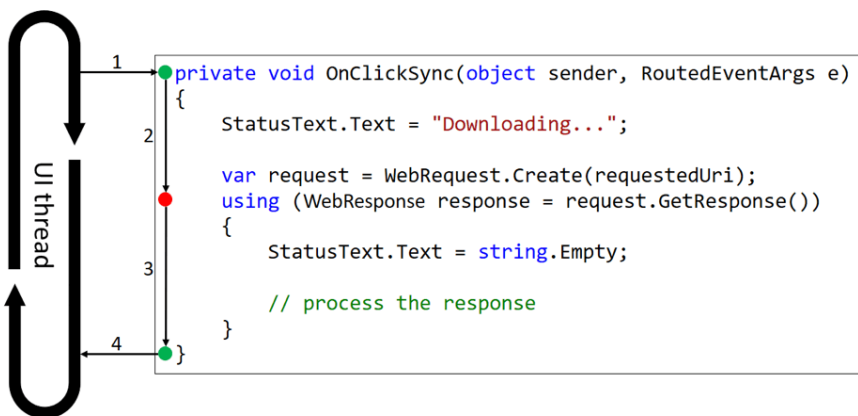


Figure 1: Code execution in a synchronous method

As shown in Figure 1,

1. the UI thread invoked the method in response to a click event (transition 1).
2. It executed the code until it reached the `GetResponse` method (marked with a red dot) which started the long running network operation (transition 2).
3. The thread was blocked there until this method completed. Only then it continued executing the code till the end of the method (transition 3).
4. Once done, the thread was released to process other events (transition 4).

From a user experience perspective, such behavior is problematic. Not only is the user not informed what is happening in the application, but also none of the users' other interactions with the application are processed during that time either.

This makes the application appear unresponsive.

When that takes too long, the operating system will show a dialog informing the user about it.

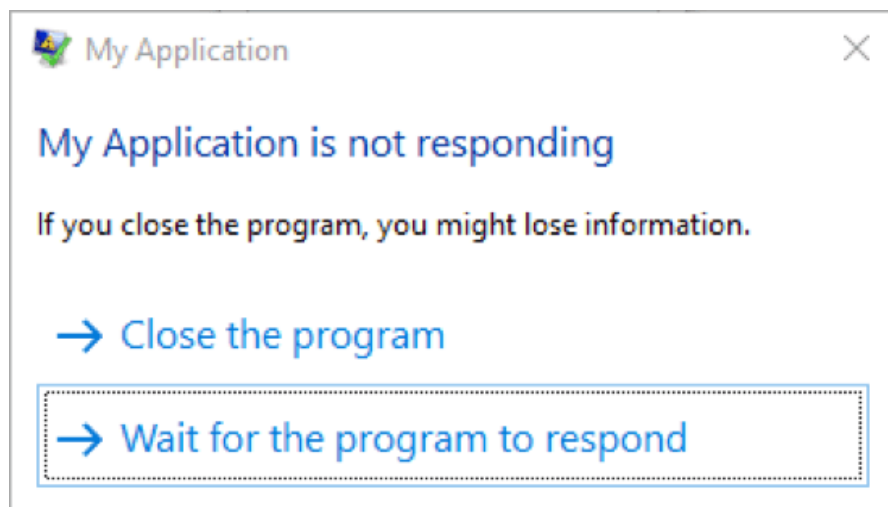


Figure 2: Windows warns about applications which stopped responding

Asynchronous programming using the `async` and `await` keywords provides a simple way to avoid this problem with minimal code changes. Here is the asynchronous version of the event handler from the previous example:

```
private async void OnClickAsync(object sender, RoutedEventArgs e)
{
    StatusText.Text = "Downloading...";

    var request = WebRequest.Create(requestedUri);
    using (WebResponse response = await request.GetResponseAsync())
    {
        StatusText.Text = string.Empty;

        // process the response
    }
}
```

Only three changes were required:

- The method signature changed from `void` to `async void`, indicating that the method is asynchronous, allowing us to use the `await` keyword in its body.
- Instead of calling the synchronous `GetResponse` method, we are calling the asynchronous `GetResponseAsync` method. By convention, asynchronous method names usually have the `Async` postfix.
- We added the `await` keyword before the asynchronous method call.

This changes how the event handler is processed:

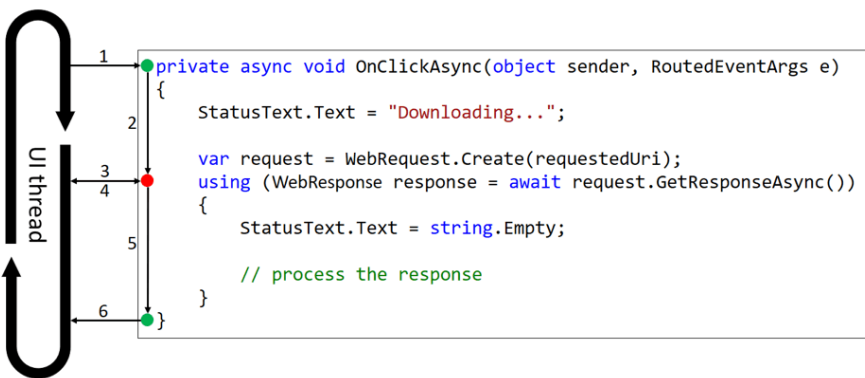


Figure 3: Code execution in an asynchronous method

As shown in Figure 3,

1. The UI thread still invokes the method in response to a click event (transition 1).
2. But now only the part of the method until the asynchronous call to the `GetResponseAsync` method (marked with a red dot) is executed synchronously (transition 2).
3. At that point, the execution of the event handler pauses (transition 3), and the UI thread can process other events (e.g. renders the status text).
4. Meanwhile, the download operation continues in the background. Once it completes, the UI thread is notified about it and will resume the execution of the event handler from the `GetResponseAsync` call onwards (transition 4).
5. It will first unwrap the method result of type `Task<WebResponse>` to `WebResponse` and assign it to the response variable. Then, it will execute the rest of the method synchronously (transition 5).
6. Once done, the UI thread will again become available for processing other events (transition 6).

Although Web applications on the server do not require their own special UI thread, they can still benefit from asynchronous programming.

A dedicated thread processes each incoming request. While this thread is busy with one request, it cannot start processing another one. Since there is a limited number of threads available in the thread pool, this limits the number of requests that can be processed in parallel. Any thread waiting for an I/O operation to complete, is therefore a wasted resource.

If the I/O operation is performed asynchronously instead, the thread is not required any more until the operation completes, and is released back to the thread pool, making it available to process other requests. Although this might slightly increase the latency of a single request, it will improve the overall throughput of the application.

67

Q67. What is the best way to start using Asynchronous methods in existing code?

Asynchronous methods can only be awaited using the `await` keyword in asynchronous methods with the `async` keyword in their signature:

```
public async Task AsyncMethod()
{
    var data = await GetDataAsync();
    // ...
}
```

In synchronous methods, without the `async` keyword in their signature, the use of the `await` keyword is not allowed and such code will not compile.

There are other ways to call asynchronous methods from synchronous ones, but they all have their own disadvantages:

- The call to the asynchronous method could be made synchronous by accessing the `Result` property or invoking the `Wait` method of its return value:

```
public void SyncMethodWithBlocking()
{
    var data = GetDataAsync().Result;
    // ...
}
```

However, this would block the invoking thread until the method returned. Preventing that is the main advantage of asynchronous methods.

- The asynchronous method could be invoked like a simple void-returning method without accessing the returned value in any way:

```
public void SyncMethodFireAndForget()
{
    GetDataAsync();
    // ...
}
```

This would be useless for methods returning a value (as in this case) because there would be no way to access that value. But even for methods not returning a value, this would mean that the execution of the calling method would continue immediately without waiting for the asynchronous method to return. Also, any exceptions thrown by the asynchronous method cannot be caught in the calling method.

Keeping this in mind, it only makes sense to call asynchronous methods from other asynchronous methods. This means that the requirement for methods to be asynchronous cascades up the call stack, all the way to an entry point method which is not explicitly called by another method in the application code. What exactly these entry point are depends on the type of the application.

In ASP.NET MVC based **web applications**, the entry points are action methods in controllers which handle incoming HTTP requests. By default, they are synchronous:

```
public void IActionResult Index()
{
    // ...
    return View();
}
```

In recent versions of ASP.NET MVC (as well as in ASP.NET Core), asynchronous action methods are fully supported. To make an action method asynchronous, only its signature needs to be changed:

```
public async Task<IActionResult> Index()
{
    var data = await GetDataAsync();
    // ...
    return View();
}
```

This is easy enough to do in any existing application. After changing the signature of any action method which needs to call an asynchronous method somewhere down its call stack, all the methods it calls until it reaches the one which has to call the asynchronous method, must also be made asynchronous.

Desktop applications (WPF and also Windows Forms) are all event driven. Their entry points are therefore event handlers. These are also synchronous by default:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // ...
}
```

Because they don't return a value, they can't return a **Task** even when made asynchronous:

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    var data = await GetDataAsync();
    // ...
}
```

Although other asynchronous methods can be awaited inside it, such an event handler method when invoked, can't be awaited by the application framework. This means that any exceptions thrown by it can't be caught directly.

To prevent the application from crashing, they must either be handled with a try/catch block in each asynchronous event handler or by a global error handler as in the

following WPF example:

```
public App()
{
    DispatcherUnhandledException += App_DispatcherUnhandledException;
}

private void App_DispatcherUnhandledException(object sender,
DispatcherUnhandledExceptionEventArgs e)
{
    var exception = e.Exception; // get exception
    // ...                       show the error to the user
    e.Handled = true;           // prevent the application from
                                crashing
}
```

So, when adding asynchronous calls to an existing application, in addition to making the event handler and all the intermediate methods in the call stack asynchronous, the error handling also needs to be revised and potentially enhanced.

In **console applications**, the only entry point is the static **Main** method:

```
static void Main(string[] args)
{
    // ...
}
```

Since C# 7.1, this method can simply be converted to an asynchronous one:

```
static async Task Main(string[] args)
{
    var data = await GetDataAsync();
    // ...
}
```

If you want the console application to exit with a return code, the asynchronous **Main** method can also return an integer value:

```
static async Task<int> Main(string[] args)
{
    var data = await GetDataAsync();
    // ...
    return result;
}
```

In older versions of C#, there's no support for the asynchronous `Main` method. You will need to write the necessary plumbing yourself:

```
static void Main(string[] args)
{
    MainAsync(args).GetAwaiter().GetResult();
}

static async Task MainAsync(string[] args)
{
    var data = await GetDataAsync();
    // ...
}
```

In this case, you'll use the new `MainAsync` method as the entry point for your code instead of the `Main` method.

Either way, after making the entry point asynchronous in an existing application, you can call asynchronous methods in it. In order to do that, all the intermediate methods must also be made asynchronous, i.e. all the methods between the new entry point method (Main or MainAsync) and the asynchronous method call you want to add.

No other changes are required to keep the application behavior unchanged.

68

Q68. What are some of the Common Pitfalls when using `async` and `await`?

The `async` and `await` keywords provide a great benefit to C# developers by making asynchronous programming easier.

Most of the times, one can use them without having to understand the inner workings in detail. At least, as long as the compiler is a good enough validator, the code will behave as intended. However, there are cases when incorrectly written asynchronous code will compile successfully, but still introduce subtle bugs that can be hard to troubleshoot and fix.

Let us look at some of the most common pitfalls.

In asynchronous methods, you should **avoid using the *async void signature*** and use `async Task` or `async Task<T>` whenever possible, where `T` is the return type of your method. The reason for this is that we should call all asynchronous methods using the `await` keyword, e.g.:

```
public async Task AsyncMethod()
{
    DoSomeStuff();                // synchronous method
    await DoSomeLengthyStuffAsync(); // long-running asynchronous
    method
    DoSomeMoreStuff();            // another synchronous method
}
```

This allows the compiler to split the calling method at the point of the `await` keyword. The first part ends with the asynchronous method call. The second part starts with using its result if any, and continues from there on.

To use the `await` keyword on a method, its return type must be `Task` or `Task<T>`. This allows the compiler to trigger the continuation of our method once the task completes. In other words, this will work if the signature of the called asynchronous method is `async Task` (or `async Task<T>`). Had the signature been `async void` instead, we would have to call it without the `await` keyword:

```
DoSomeStuff();                // synchronous method
DoSomeLengthyStuffAsync();    // long-running asynchronous method
DoSomeMoreStuff();            // another synchronous method
```

The compiler would not complain, though.

If subsequent code doesn't depend on the side effects of `DoSomeLengthyStuffAsync`, it might even work correctly.

However, there is one important difference between the two examples.

In the first one, `DoSomeMoreStuff` will only be invoked after `DoSomeLengthyStuffAsync` completes.

In the second one, `DoSomeMoreStuff` will be invoked immediately after `DoSomeLengthyStuffAsync` starts.

Since in the latter case, `DoSomeLengthyStuffAsync` and `DoSomeMoreStuff` run in parallel, race conditions might occur.

If `DoSomeMoreStuff` depends on any of `DoSomeLengthyStuffAsync`'s side effects, these might or might not yet be available when `DoSomeMoreStuff` wants to use them. Such

a bug can be difficult to fix because it cannot be reproduced reliably. It might even occur only in production environment where I/O operations are usually slower than in development environment.

To avoid such issues altogether, always use the `async Task` as the signature for methods you intend to call from your code. Restrict the usage of the `async void` signature to event handlers, which are not allowed to return anything:

```
private async void OnEventRaised(object sender, EventArgs e)
{
    // ...
}
```

Make sure you never call these event handler methods yourself. If you need to reuse the code of an event handler, refactor it into a separate method returning `Task`, and call that new method from both the event handler and your method, using `await`.

In a way, asynchronous methods behave contagiously. To call an asynchronous method with `await`, you must make the calling method asynchronous as well, even if it was not `async` earlier. Now, all methods calling this newly asynchronous method must also become asynchronous. This pattern repeats itself up the call stack until it finally reaches the entry points, e.g. event handlers.

When one of the methods on this path to the entry points cannot be asynchronous, this poses a problem. For example, constructors. They cannot be asynchronous; therefore, you cannot use `await` in their body. You could break the asynchronous requirement early by giving a method `async void` signature, but this prevents you from waiting for its execution to end, which makes it a bad idea in most cases.

Alternatively, you could try synchronously waiting for the asynchronous method to complete, by calling the `Wait` method on the returned `Task`, or reading its `Result` property. Of course, this synchronous code will temporarily block the calling thread which we wanted to avoid in the first place.

In a desktop application, this would be the UI thread. Blocking it would make the application unresponsive to user input. Even worse, in some cases you could cause a **deadlock** in your application with some very innocent looking code:

```
private void MyEventHandler(object sender, RoutedEventArgs e)
{
    var instance = new InnocentLookingClass();
    // further code
}
```

Any synchronously called asynchronous code in `InnocentLookingClass` constructor is enough to cause a deadlock:

```
public class InnocentLookingClass {
    public InnocentLookingClass()
    {
        DoSomeLengthyStuffAsync().Wait();
        // do some more stuff
    }

    private async Task DoSomeLengthyStuffAsync()
    {
        await SomeOtherLengthyStuffAsync();
    }

    // other class members
}
```

Let us dissect what is happening in this code.

`MyEventHandler` synchronously calls `InnocentLookingClass` constructor, which invokes `DoSomeLengthyStuffAsync`, which in turn asynchronously invokes `SomeOtherLengthyStuffAsync`. The execution of the latter method starts. At the same time, the UI thread blocks at the `Wait` call until `DoSomeLengthyStuffAsync` completes, not being able to process any incoming events.

Eventually `SomeOtherLengthyStuffAsync` completes and notifies the UI thread that the execution of `DoSomeLengthyStuffAsync` can continue. Unfortunately, the UI thread is waiting for that method to complete instead of processing new events, and will therefore never be triggered to continue. Hence, it will wait indefinitely.

As you can see, synchronously invoking asynchronous methods can quickly have undesired consequences. Avoid it at all costs, unless you are sure what you are doing,

e.g. you are not blocking the UI thread in a desktop application.

The deadlock in the above example would not happen if `DoSomeLengthyStuffAsync` did not require to be continued on the UI thread where it was running before the asynchronous call. In this case, it would not matter that this thread was busy waiting for it to complete, and the **execution could continue on another thread**. Once completed, the constructor execution could continue as well.

As it turns out, there is a way to achieve this when awaiting asynchronous calls – by invoking `ConfigureAwait(false)` on the returned `Task` before awaiting it:

```
await SomeOtherLengthyStuffAsync().
ConfigureAwait(continueOnCapturedContext: false);
```

This modification would avoid the deadlock, although the problem of the synchronous call in the constructor blocking the UI thread until it completes, would remain.

While allowing continuation on a different thread in the above example might not be the best approach, there are scenarios in which it makes perfect sense. Continuing execution on the originating thread affects performance, and as long as you are sure that none of the code after the call to the awaited asynchronous method needs the original execution context to be restored, disabling it will make your code run faster.

You might wonder - which code requires the context to be restored? This depends on the type of the application:

- In user-interface based applications (Windows Forms, WPF and UWP), any code that interacts with user interface components must run on the UI thread.
- In web applications (ASP.NET and ASP.NET Core), a thread pool thread with the calling context restored is required for any code accessing the request context or authentication information.

When you are unsure, you can use the following rule of thumb, which works fine in most cases:

- Code in reusable class libraries can safely disable context restoration.
- Application code should keep the default continuation on the originating thread – just to be on the safer side.

69

Q69. How are Exceptions handled in Asynchronous code?

The Task Parallel Library (TPL) extensively uses the `AggregateException` exception to model the fact that asynchronous operations can throw more than one exception. All these exceptions can be accessed and inspected through its `InternalExceptions` property.

It's important to be aware of this when handling exceptions in the TPL because it means that an `AggregateException` exception will be thrown instead of the actual exception thrown inside the task.

An exception which was not caught in the task will be thrown from it when its `Wait` method is called:

```
try
{
    ThrowsNotImplementedExceptionAsync().Wait();
}
catch (AggregateException e) when (e.InnerExceptions[0] is
    NotImplementedException inner)
```

```
{
    // handle inner exception
}
```

For tasks which return results, accessing their **Result** property will also cause any uncaught exceptions to be thrown wrapped into an **AggregateException** exception:

```
try
{
    var result = ThrowsNotImplementedExceptionAsync(42).Result;
}
catch (AggregateException e) when (e.InnerExceptions[0] is
    NotImplementedException inner)
{
    // handle inner exception
}
```

As demonstrated by the sample code, the actual exception thrown (**NotImplementedException** in the case of the **ThrowsNotImplementedExceptionAsync** methods) is stored in the **InnerExceptions** collection.

It's also a good practice to follow these exception handling patterns when implementing methods which return a **Task** or a **Task<T>**. The consuming code will most likely expect the exceptions from such a method to be wrapped into an **AggregateException** exception. Therefore, you should avoid throwing exceptions directly from such methods:

```
Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        // not recommended
        throw new ArgumentOutOfRangeException(nameof(input));
    }

    return Task.Run(() => Process(input));
}
```

When an exception is thrown from a delegate being executed by a task (instead

of directly as in the above example), it automatically gets wrapped into an **AggregateException** exception by the TPL infrastructure code. Under the hood, a task is put into a faulted state when an exception is thrown from the delegate it is executing. There's a convenient helper method available to create a faulted task from an exception without having to run a task. You should use that method instead of throwing the exception directly:

```
Task<int> ImprovedProcessAsync(int input)
{
    if (input < 0)
    {
        return Task.FromException<int>(new
            ArgumentOutOfRangeException(nameof(input)));
    }

    return Task.Run(() => Process(input));
}
```

The **async** and **await** keywords were introduced to make asynchronous code seem more similar to synchronous code. An important part of that is also the exception handling. The thrown **AggregateException** exceptions are automatically unwrapped back to the original type that was thrown inside the task:

```
try
{
    await ThrowsNotImplementedExceptionAsync();
}
catch (NotImplementedException e)
{
    // handle exception
}
```

It's important to notice that the exception from the task is thrown at the point of the **await** keyword. This means that if the task is not awaited but executed in a "fire and forget" manner without the **await** keyword, any uncaught exceptions from the task will not be thrown in the calling method:


```
try
{
    ThrowsNotImplementedExceptionAsync();
}
catch (Exception e)
{
    // won't catch exception
}
```

The difference of behavior between using the Task Parallel Library with and without the `async` and `await` keywords is even greater when dealing with multiple tasks. Typically, these will be collected in an array which can then be passed to the static helper methods on the `Task` class:

```
var tasks = new[]
{
    ThrowsNotImplementedExceptionAsync(),
    ThrowsNotImplementedExceptionAsync()
};
```

To wait for all the tasks to complete without using `async` and `await`, the `Task.WaitAll` method can be used:

```
try
{
    Task.WaitAll(tasks);
}
catch (AggregateException e)
{
    foreach (var inner in e.InnerExceptions)
    {
        // handle exception
    }
}
```

The behavior of the `Task.WaitAll` method is very similar to the `Wait` method on the task instance. Any exceptions thrown in the tasks passed to it will be wrapped in an `AggregateException` exception thrown by the `WaitAll` method. They can be inspected by iterating through the `InnerExceptions` collection.

To wait for multiple tasks asynchronously using the `await` keyword, the `Task.WhenAll` method must be used instead. To make the call more similar to synchronous code, the thrown exceptions again aren't wrapped into an `AggregateException` exception:

```
try
{
    await Task.WhenAll(tasks);
}
catch (NotImplementedException e)
{
    // handle exception

    foreach (var task in tasks)
    {
        if (task.IsFaulted)
        {
            // handle task.Exception
        }
    }
}
```

However, this means that only one of the potentially multiple exceptions not caught in the tasks will be thrown from the awaited `Task.WhenAll` method. If we want to inspect all of them, we must access each of the tasks directly. If a task has thrown an uncaught exception, its `IsFaulted` property will be set to `true`. The exception thrown is stored in its `Exception` property.

Perhaps even more surprising is the behavior of the blocking `Task.WaitAny` method which only waits for one of the tasks passed to it to complete, not all of them:

```
try {
    var completedIndex = Task.WaitAny(tasks);

    if (tasks[completedIndex].IsFaulted)
    {
        // handle task.Exception
    }
}
```

```

catch (Exception e)
{
    // won't catch exception
}

```

What you might not expect is that the `Task.WaitAny` method will not throw any exceptions from the tasks it waited for. To see if an uncaught exception was thrown by the completed task, it needs to be inspected directly. I use the index of the completed task returned by the method to access the correct task, and check if it has thrown an exception.

The `Task.WhenAny` method intended for use with the `await` keyword exhibits similar behavior:

```

try
{
    var task = await Task.WhenAny(tasks);

    if (task.IsFaulted)
    {
        // handle task.Exception
    }
}
catch (NotImplementedException e)
{
    // won't catch exception
}

```

It doesn't throw any exceptions for the awaited tasks, but it returns the task which has completed. This task can then be inspected directly whether it has completed successfully or has thrown an exception.

70

Q70. What are some Best Practices for Asynchronous file operations?

In software, I/O operations usually take hundreds or even thousand times longer than computation.

File operations are the most common I/O operations. Although nowadays files are usually read locally and from fast SSD (solid state drive) disks, the operation can still take a noticeably longer time, especially for larger files.

Until the introduction of asynchronous programming with `async` and `await`, the easiest way to implement file operations was synchronously:

```
private string ReadAllText(string filepath)
{
    var buffer = new byte[bufferSize];
    var stringBuilder = new StringBuilder();

    using (var stream = new FileStream(filepath, FileMode.Open))
    {
        int bytesRead;
```

```

while ((bytesRead = stream.Read(buffer, 0, buffer.Length)) > 0)
{
    stringBuilder.Append(Encoding.ASCII.GetString(buffer, 0,
        bytesRead));
}
return stringBuilder.ToString();
}
}

```

A negative side effect of synchronous file operations is that the thread can't do anything else for the duration of the operation.

For desktop applications, this means that they will not respond to user interactions. For small files, this might not be a problem. But once the duration of the operation exceeds a few hundred milliseconds, the user will start noticing the unresponsiveness and wonder what's wrong with the application.

To avoid this issue, file operations should always be performed asynchronously. With **async** and **await**, this piece of code won't be any more complicated than the synchronous version:

```

private async Task<string> ReadAllTextAsync(string filepath)
{
    var buffer = new byte[bufferSize];
    var stringBuilder = new StringBuilder();

    using (var stream = new FileStream(filepath, FileMode.Open))
    {
        int bytesRead;
        while ((bytesRead = await stream.ReadAsync(buffer, 0, buffer.
            Length)) > 0)
        {
            stringBuilder.Append(Encoding.ASCII.GetString(buffer, 0,
                bytesRead));
        }
        return stringBuilder.ToString();
    }
}

```

However, during an asynchronous file operation, the application will remain responsive to user interaction.

Ever since asynchronous programming was introduced with `async` and `await` in .NET framework 4.5, asynchronous versions of many APIs for file operations were added. The `ReadAsync` method from the sample we just saw, is one such example. It's the asynchronous version of the `Read` method from the first example.

Unless you don't mind blocking a thread for a file operation, before using a synchronous API, you should always look for an asynchronous API. If there isn't one, you should try to find an asynchronous alternative (typically a lower level one) which you can use to achieve the same result.

In comparison to synchronous methods, asynchronous methods often have additional parameters for extra functionalities. The most common one is the `CancellationToken` parameter:

```
private async Task<string> ReadAllTextAsync(string filepath,
CancellationToken cancellationToken)
{
    var buffer = new byte[bufferSize];
    var stringBuilder = new StringBuilder();

    using (var stream = new FileStream(filepath, FileMode.Open))
    {
        int bytesRead;
        while ((bytesRead = await stream.ReadAsync(buffer, 0, buffer.
            Length, cancellationToken) ) > 0)
        {
            stringBuilder.Append(Encoding.ASCII.GetString(buffer, 0,
                bytesRead));
        }
        return stringBuilder.ToString();
    }
}
```

The `CancellationTokenSource` class is usually used to create a cancellation token:

```

public partial class MainWindow : Window
{
    private CancellationTokenSource cancellationTokenSource = null;

    // ...

    private async void OnReadFileClicked(object sender,
        RoutedEventArgs e)
    {
        using (cancellationTokenSource = new CancellationTokenSource())
        {
            var text = await ReadAllTextAsync(path,
                cancellationTokenSource.Token);
        }
        cancellationTokenSource = null;
    }
}

```

By passing a cancellation token to an asynchronous method, you get the ability to cancel the asynchronous operation before it completes:

```

private void OnCancelClicked(object sender, RoutedEventArgs e)
{
    if (cancellationTokenSource != null)
    {
        cancellationTokenSource.Cancel();
    }
}

```

The second standard additional parameter of asynchronous methods is `IProgress<T>`. It is used to allow the asynchronous method to report its progress during the operation. Unfortunately, unlike the `CancellationToken` parameter which is supported by almost all asynchronous methods in the .NET framework, the `IProgress<T>` is rarely supported.

In spite of that, it's a good idea to support the parameter in your own asynchronous methods when you need to implement progress reporting:

```

private async Task<string> ReadAllTextAsync(string filepath,
IPProgress<int> progress, CancellationToken cancellationToken)
{
    var fileInfo = new FileInfo(filepath);
    var fileSize = (int)fileInfo.Length;
    var buffer = new byte[bufferSize];
    var stringBuilder = new StringBuilder();

    using (var stream = new FileStream(filepath, FileMode.Open))
    {
        int bytesRead;
        while ((bytesRead = await stream.ReadAsync(buffer, 0, buffer.
            Length, cancellationToken)) > 0)
        {
            stringBuilder.Append(Encoding.ASCII.GetString(buffer, 0,
                bytesRead));
            progress.Report(stringBuilder.Length * 100 / fileSize);
        }
        return stringBuilder.ToString();
    }
}

```

For the sake of simplicity, I report the progress as an int value between 0 and 100. Depending on the operation, a more complex structure can be used to describe the progress more precisely. In this case, it could for example contain the total size of the file and the number of bytes read so far.

To invoke a method with an `IPProgress<T>` parameter, an instance of a class implementing this interface must be passed to it. You can always create your own, but there's also one built into the .NET framework – `Progress<T>`:

```

using (cancellationTokenSource = new CancellationTokenSource())
{
    var progress = new Progress<int>(percent => ProgressBar.Value =
        percent);
    var text = await ReadAllTextAsync(path, progress,
        cancellationTokenSource.Token);
}

```


Its constructor accepts a delegate which will be invoked whenever a new progress value is reported by the asynchronous method. As a bonus, the delegate will automatically be invoked on the UI thread when in a desktop application. That's why I can directly assign a value to the ProgressBar component in the sample above without using a dispatcher to access the component from the UI thread.

In the samples throughout the chapter, I was implementing a method for reading an entire file as a string.

This method is already implemented as a static method on the `System.IO.File` class.

In the .NET framework, there's only a synchronous version available. In .NET Core, there's also an asynchronous version available since version 2.0. However, there is no overload with an `IProgress<T>` parameter.

71

Q71. What is a Critical Section and how to correctly implement it?

In multithreaded programming, a **critical section** is a block of code which shouldn't be entered by more than one thread at the same time, otherwise incorrect behavior or data inconsistency might occur.

Typically, such blocks of code access a shared resource (e.g. an internal data structure or an external file or network connection). If multiple threads modify that shared resource in a specific manner, they can corrupt the data in it.

The following method for withdrawing an amount of money from a bank account is an example of a code which must be implemented as a critical section:

```
public bool Withdraw(decimal amount)
{
    if (amount > Balance)
    {
        return false;
    }
}
```

```

    Balance -= amount;
    return true;
}

```

Before actually withdrawing the amount, the method checks if the balance on the account is high enough to avoid a negative balance on the account after the withdrawal.

Now, imagine that two different threads want to withdraw all the remaining money from the account, at the same time. If they both manage to check the balance before the other thread has withdrawn the amount, they will both pass the check and therefore also withdraw the money.

However, at the end, the balance on the account will be negative:

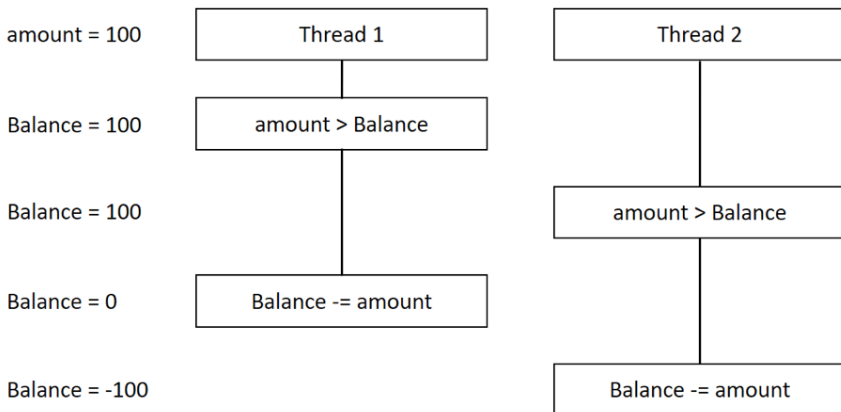


Figure 1: Two withdrawals in parallel cause inconsistency

To prevent such inconsistencies, the `Withdraw` method should be implemented as a critical section and should allow only one thread to enter it at a time. If a thread attempts to enter the method while another thread is already executing the code in it, it should wait until that first thread is done.

In C#, the `lock` statement can be used to implement a critical section:

```

private readonly object lockObject = new object();

public bool Withdraw(decimal amount)
{

```

```
lock (lockObject)
{
    if (amount > Balance)
    {
        return false;
    }

    Balance -= amount;
    return true;
}
```

When a thread enters the lock block of code, a lock is established on the `lockObject`. When another thread tries to enter the same block of code, it is blocked if there is already a lock on the `lockObject`.

When the first thread exits the lock block, it releases the lock on the `lockObject`. This allows the second thread to enter the lock block and establish its own lock on the `lockObject` to prevent other threads entering the lock block.

With this protection in place, the balance on the bank account can't change any more between the check and the actual withdrawal. **This ensures data consistency.**

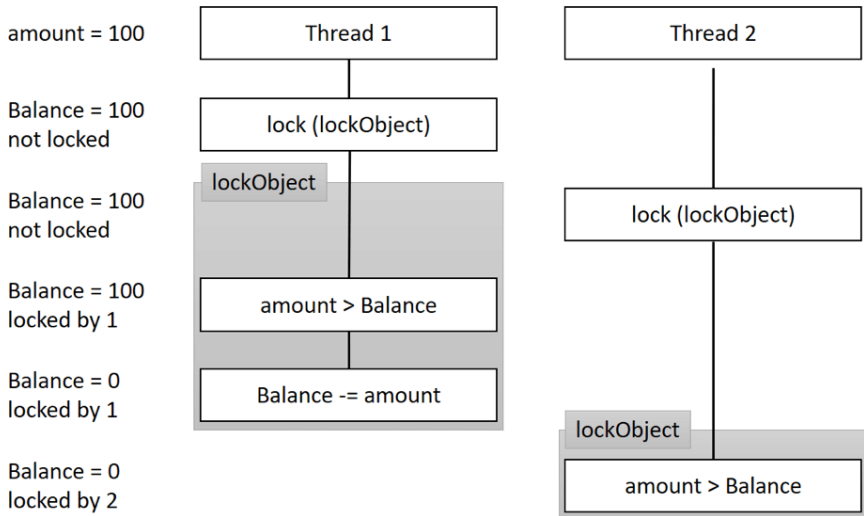


Figure 2: Critical section prevents data inconsistency

It is important that the critical section owns and has full control of the lock object it uses.

Since any reference type value can serve as a lock object, one might be tempted to avoid creating a separate dedicated object and just use another already existing reference type value. In the case of the `Withdraw` method, this could for example be the `BankAccount` object containing the method:

```
public bool Withdraw(decimal amount)
{
    lock (this)
    {
        if (amount > Balance)
        {
            return false;
        }

        Balance -= amount;
        return true;
    }
}
```

In a way, this seems to make perfect sense. We are locking the `BankAccount` object because we are protecting its data. But because the `BankAccount` object can also be accessed by other code, nothing is preventing that code from also obtaining locks on the same object as in the following contrived example:

```
private void Transfer(decimal amount, BankAccount from, BankAccount to)
{
    lock (to)
    {
        from.Withdraw(amount);
        to.Deposit(amount);
    }
}
```

If we're using the `Withdraw` method which locks the `BankAccount` object, the `Transfer` method above might cause problems.

Let's imagine that two threads invoke it, trying to transfer some amount in the opposite direction. They could cause a deadlock, i.e. reach a situation in which each thread is holding a lock on one bank account and waiting for the lock on the other bank account to be released:

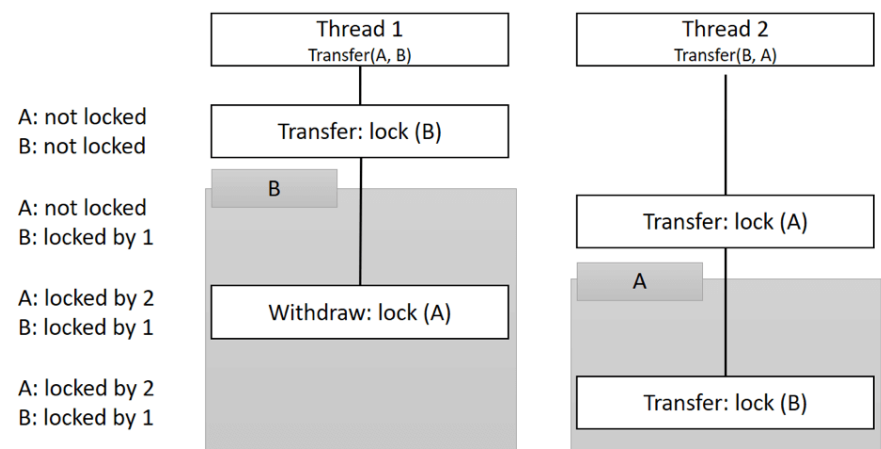


Figure 3: Deadlock caused by locking the same object in different contexts

The **Withdraw** method which creates a lock on its own internal lock object would avoid this issue. In that case, each of the four method invocations would lock a different object and the threads would not have to wait for existing locks to be released.

That's one of the reasons why each lock object should only be used for its own well-defined scenario.

It could still be in two different methods if they both access the same resource. The important part is that the lock object should not be accessible to external code that's not under our control, which could otherwise misuse it and cause problems.

72

Q72. How to manage multiple threads in the .NET framework?

Although you can achieve some rudimentary synchronization between threads by awaiting tasks and using utility methods such as `Task.WhenAll` and `Task.WhenAny`, this doesn't allow any sophisticated interactions. When these are required, different synchronization primitives in the .NET framework will be the right choice.

The simplest one of them is the `lock` statement which ensures that multiple threads don't enter a block of code at the same time:

```
private readonly object lockObject = new object();
public bool WithdrawLock(decimal amount)
{
    lock (lockObject)
    {
        if (amount > Balance) // check available amounts
        {
            return false;
        }
    }
}
```

```
        Balance -= amount; // withdraw money
        return true;
    }
}
```

When a thread enters the lock block in the sample above, it acquires a lock on the **lockObject**. When a different thread reaches the lock statement, it will be blocked until that lock is released. This will prevent the **Balance** value from changing between the check of the available funds and the actual withdrawal.

The lock statement is just a shorthand for using the **Monitor** class:

```
private readonly object lockObject = new object();

public bool Withdraw(decimal amount)
{
    var lockAcquired = false;
    try
    {
        Monitor.Enter(lockObject, ref lockAcquired);

        if (amount > Balance)
        {
            return false;
        }

        Balance -= amount;
        return true;
    }
    finally
    {
        if (lockAcquired)
        {
            Monitor.Exit(lockObject);
        }
    }
}
```

In the call to the **Monitor.Enter** method, the value of **lockAcquired** will remain set to **false** if an error happens. If no exception is thrown while the method is waiting

(e.g. because of a `Thread.Interrupt` call), the lock will always be successfully acquired when the method completes.

Of course, there's no reason for using the longer syntax to achieve the same functionality. You would only run into the risk of not implementing the code properly and could introduce a bug. However, the `Monitor` class provides additional features which are not available through the lock statement.

For example, a timeout can be specified for acquiring the lock:

```
private readonly object lockObject = new object();

public bool Withdraw(decimal amount)
{
    if (Monitor.TryEnter(lockObject, timeout))
    {
        try
        {
            if (amount > Balance)
            {
                return false;
            }

            Balance -= amount;
            return true;
        }
        finally
        {
            Monitor.Exit(lockObject);
        }
    }
    else
    {
        // lock wasn't acquired
        return false;
    }
}
```

If the lock can't be acquired in the given amount of time, the method call will return false and the execution will continue. This approach can be used to limit the

maximum wait time and avoid infinite waits when a deadlock happens.

The **Mutex** class is a more heavyweight version of the **Monitor** class that relies on the underlying operating system. This allows it to synchronize access to a resource not only across thread boundaries, but even over process boundaries. **Monitor** is the recommended alternative over **Mutex** for synchronization inside a single process.

The **SemaphoreSlim** and **Semaphore** classes can be used when you need to limit the number of concurrent consumers of a resource to a configurable maximum number, instead of to only a single one, as with the **Monitor** class.

```
private void Process(SemaphoreSlim semaphore)
{
    semaphore.Wait();
    try
    {
        // do the processing
    }
    finally
    {
        semaphore.Release();
    }
}

var semaphore = new SemaphoreSlim(3, 3);

var tasks = Enumerable.Range(0, 10)
    .Select(index => Task.Run(() => Process(semaphore)))
    .ToArray();

Task.WaitAll(tasks);
```

The code above will never have more than three tasks doing the processing at the same time. The **Wait** method of the semaphore will block after three calls (the remaining seven tasks will be blocked). Only a call to the **Release** method will make a new slot available.

The constructor of the **SemaphoreSlim** class is a bit confusing to use. The second parameter specifies how many consumers at most can be active at the same time. The first parameter specifies how many of those available slots are still free when the

constructor is called. In the code sample we just saw, three is the number of allowed concurrent consumers, and all three of those slots are still available at the start.

`SemaphoreSlim` is more lightweight than `Semaphore` but restricted to only a single process. Whenever possible, you should use `SemaphoreSlim` instead of `Semaphore`.

The `ReaderWriterLockSlim` class can differentiate between two different types of access to a resource. It allows unlimited number of readers to access the resource in parallel, and limits writers to a single access at a time. It is great for protecting resources that are thread safe for reading but require exclusive access for modifying data. It could for example be used for simple protection of a dictionary for multithreaded use:

```
public class DictionaryWithReaderWriterLock<TKey, TValue>
{
    private readonly ReaderWriterLockSlim dictionaryLock = new
        ReaderWriterLockSlim();
    private readonly Dictionary<TKey, TValue> dictionary;

    public DictionaryWithReaderWriterLock()
    {
        dictionary = new Dictionary<TKey, TValue>();
    }

    public TValue this[TKey key]
    {
        get
        {
            dictionaryLock.EnterReadLock();
            try
            {
                return dictionary[key];
            }
            finally
            {
                dictionaryLock.ExitReadLock();
            }
        }
        set
        {

```

```
        dictionaryLock.EnterWriteLock();
    try
    {
        dictionary[key] = value;
    }
    finally
    {
        dictionaryLock.ExitWriteLock();
    }
}

public void Add(TKey key, TValue value)
{
    dictionaryLock.EnterWriteLock();
    try
    {
        dictionary.Add(key, value);
    }
    finally
    {
        dictionaryLock.ExitWriteLock();
    }
}
```

Any number of concurrent read accesses to a dictionary are thread safe. But when modifying a dictionary, no other readers or writers are allowed.

`AutoResetEvent`, `ManualResetEvent` and `ManualResetEventSlim` will block incoming threads, until they receive a signal (i.e. a call to the `Set` method). Then the waiting threads will continue their execution:

```
var handle = new AutoResetEvent(false);

var task = Task.Run(() =>
{
    Console.WriteLine("Task start");
    handle.WaitOne();
});
```

```

    Console.WriteLine("Task end");
});

Thread.Sleep(10);

Console.WriteLine("Before signal");
handle.Set();
Console.WriteLine("After signal");

task.Wait();

```

When the above code is executed, it will produce the following output:

```

Task start
Before signal
After signal
Task end

```

The call to the **Thread.Sleep** method will give the newly created task enough time to start executing before the current thread continues. But when the task reaches the call to the **WaitOne** method, its execution will be blocked until the **Set** method is called from the main thread.

AutoResetEvent will only allow one thread to continue. Then it will block again until the next call to the **Set** method. **ManualResetEvent** and **ManualResetEventSlim** will remain in the set state (non-blocking), until the **Reset** method is called.

Being more lightweight, **ManualResetEventSlim** is the recommended version of the two.

Choosing the right synchronization class for the job can greatly simplify the implementation of correct interaction between multiple threads. It will often also improve performance because threads will never be waiting unless absolutely necessary, to ensure data consistency.

SECTION VIII

SERIALIZATION & REFLECTION

73

Q73. What types of Serialization are supported in the .NET framework?

Serialization is used to convert in-memory instances of types into a stream of bytes. This byte stream can then be persisted to storage or transferred to another process running on either the same machine or another computer.

Deserialization is a reverse process, converting that stream of bytes back into an instance of a type.

The .NET framework has built-in support for several different types of serialization. The recommended one is **data contract serialization**. It originates from [WCF \(Windows Communication Foundation\)](#) which uses it to serialize arguments in its requests, and return values in its responses.

To use data contract serialization, types must be annotated with the **DataContract** attribute. Only properties and fields annotated with the **DataMember** attribute will be included in serialization. Both public and non-public members can be serialized:

```
[DataContract]
public class Person {
    [DataMember]
    public string FirstName { get; set; }
    [DataMember]
    public string LastName { get; set; }
    public int Age
    {
        get
        {
            var today = DateTime.Today;
            var age = today.Year - dateOfBirth.Year;
            if (dateOfBirth.AddYears(age) > today)
            {
                age--;
            }
            return age;
        }
    }
    [DataMember]
    protected DateTime dateOfBirth;

    public Person(string firstName, string lastName, DateTime
dateOfBirth)
    {
        FirstName = firstName;
        LastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }
}
```

To serialize an instance of a type, a serializer for that type must first be instantiated:

```
var person = new Person("John", "Doe", new DateTime(1990, 1, 1));
var serializer = new DataContractSerializer(typeof(Person));
using (var stream = File.Create(path))
{
    serializer.WriteObject(stream, person);
}
```


This will produce an XML representation of the serialized instance:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/
Serialization.DataContract" xmlns:i="http://www.w3.org/2001/
XMLSchema-instance">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <dateOfBirth>1990-01-01T00:00:00</dateOfBirth>
</Person>
```

The same serializer instance can be used for deserialization. The returned object must be manually cast to the correct type:

```
using (var stream = File.OpenRead(path))
{
    var deserialized = (Person)serializer.ReadObject(stream);
}
```

Data contract serialization will automatically serialize the complete object graph consisting of different types referenced through properties and fields. By default, it can't handle cycles in the graph as in the following example of nodes for a [doubly linked list](#):

```
[DataContract]
public class Node<T>
{
    [DataMember]
    public T Value { get; set; }
    [DataMember]
    public Node<T> Next { get; set; }
    [DataMember]
    public Node<T> Previous { get; set; }
}

var firstNode = new Node<int> { Value = 1 };
var secondNode = new Node<int> { Value = 2 };
firstNode.Next = secondNode;
secondNode.Previous = firstNode;
```

Trying to serialize either node will throw a `SerializationException` exception. However, the serializer can be configured to keep track of object references in the serialization process:

```
var settings = new DataContractSerializerSettings()
{
    PreserveObjectReferences = true
};

var serializer = new DataContractSerializer(typeof(Node<int>),
settings);
serializer.WriteObject(stream, firstNode);
```

In this mode, it assigns an `Id` value to each instance and uses it as a reference when the same instance reappears in the object graph:

```
<NodeOfint z:Id="1" xmlns="http://schemas.datacontract.org/2004/07/
Serialization.DataContract" xmlns:i="http://www.w3.org/2001/
XMLSchema-instance" xmlns:z="http://schemas.microsoft.com/2003/10/
Serialization/">
  <Next z:Id="2">
    <Next i:nil="true"/>
    <Previous z:Ref="1" i:nil="true"/>
    <Value>2</Value>
  </Next>
  <Previous i:nil="true"/>
  <Value>1</Value>
</NodeOfint>
```

Data contract serialization also needs help when the serialized type includes interfaces or abstract classes. All supported derived types must be declared in advance using the `KnownTypes` attribute:

```
[DataContract]
[KnownType(typeof(Bow))]
public class Hero {
    [DataMember]
    public IWeapon Weapon { get; set; }
}

public interface IWeapon
```

```

{
    int Damage { get; set; }
}

[DataContract]
public class Bow : IWeapon
{
    [DataMember]
    public int Damage { get; set; }
    [DataMember]
    public int Arrows { get; set; }
}

```

The serializer doesn't need any additional configuration in this case:

```

var serializer = new DataContractSerializer(typeof(Hero));
serializer.WriteObject(stream, hero);

```

The serialized XML will include information about the actual type which will be used during deserialization:

```

<Hero xmlns="http://schemas.datacontract.org/2004/07/Serialization.
DataContract" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
    <Weapon i:type="Bow">
        <Arrows>10</Arrows>
        <Damage>20</Damage>
    </Weapon>
</Hero>

```

Data contract serialization doesn't allow full control over the shape of the generated XML. When that is required, **XML serialization** should be used instead. In its basic form, the types to serialize don't require any additional annotations:

```

public class Person {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}

```

The serializer instance is again type specific:

```
var serializer = new XmlSerializer(typeof(Person));
serializer.Serialize(stream, person);
```

The generated XML will use the type and member names as element names:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <DateOfBirth>1990-01-01T00:00:00</DateOfBirth>
</Person>
```

The same serializer instance can also be used for deserialization:

```
var deserialized = (Person)serializer.Deserialize(stream);
```

Attributes can be used for more control over the generated XML:

```
[XmlRoot("person")]
public class PersonWithAttributes
{
    [XmlElement("name")]
    public string FirstName { get; set; }
    [XmlIgnore]
    public string LastName { get; set; }
    [XmlAttribute("dateOfBirth")]
    public DateTime DateOfBirth { get; set; }
}
```

In the example we just saw, I used attributes to change the name of elements, to serialize members as XML attributes instead of elements, and to exclude a member from serialization. More attributes [are available](#).

However, circular references in the object graph are not supported because the serializer has no concept of references. It fully serializes an instance on each encounter even if it is referenced multiple times in the object graph.

XML serialization also offers support for [XML schemas](#). Using the `xsd.exe` command

line tool, a schema can be generated for a type:

```
xsd MyAssembly.dll /type:PersonWithAttributes
```

The result will be output into an .xsd file:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.
w3.org/2001/XMLSchema">
  <xs:element name="person" nillable="true"
type="PersonWithAttributes" />
  <xs:complexType name="PersonWithAttributes">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" name="name"
type="xs:string" />
    </xs:sequence>
    <xs:attribute name="dateOfBirth" type="xs:dateTime"
use="required" />
  </xs:complexType>
</xs:schema>
```

The tool also works in reverse. It can generate a class (or multiple classes) to use for serialization and deserialization according to a provided XML schema:

```
xsd PersonWithAttributes.xsd /classes
```

This will generate a partial class with properties matching the schema:

```
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd",
"4.6.1055.0")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlRootAttribute("person", Namespace="",
IsNullable=true)]
public partial class PersonWithAttributes {
```

```
private string nameField;
private System.DateTime dateOfBirthField;
/// <remarks/>
public string name {
    get {
        return this.nameField;
    }
    set {
        this.nameField = value;
    }
}

/// <remarks/>
[System.Xml.Serialization.XmlAttributeAttribute()]
public System.DateTime dateOfBirth {
    get {
        return this.dateOfBirthField;
    }
    set {
        this.dateOfBirthField = value;
    }
}
}
```

The final serialization type in the .NET framework is **binary serialization**. As the name already implies, this one doesn't produce an XML output. Serialized objects are represented as a binary stream of bytes with no textual representation.

For a type to be serializable with binary serialization, it must be annotated with the **Serializable** attribute:

```
[Serializable]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age
    {
        get
```

```

{
    var today = DateTime.Today;
    var age = today.Year - dateOfBirth.Year;
    if (dateOfBirth.AddYears(age) > today)
    {
        age--;
    }
    return age;
}
}
protected DateTime dateOfBirth;

public Person(string firstName, string lastName, DateTime
dateOfBirth)
{
    FirstName = firstName;
    LastName = lastName;
    this.dateOfBirth = dateOfBirth;
}
}

```

This serialization is based on reflection. It automatically includes the full class and assembly name along with all (public and non-public) properties and fields. Because of reflection, the serializer is not type specific and the same instance can be used for serialization and deserialization of all types:

```

var formatter = new BinaryFormatter();

using (var stream = File.Create(path))
{
    formatter.Serialize(stream, person);
}

using (var stream = File.OpenRead(path)) {
    var deserialized = (Person)formatter.Deserialize(stream);
}

```

The serializer can handle full object graphs (as long as all types in the graph are annotated with the `Serializable` attribute), including circular references.

To exclude a field or property from serialization (e.g. because it contains sensitive or redundant data), it must be annotated with the **NonSerialized** attribute:

```
[Serializable]
public class PersonCustomized
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age
    {
        get
        {
            var today = DateTime.Today;
            var age = today.Year - dateOfBirth.Year;
            if (dateOfBirth.AddYears(age) > today)
            {
                age--;
            }
            return age;
        }
    }
    protected DateTime dateOfBirth;
    [NonSerialized]
    private string password;

    public PersonCustomized(string firstName, string lastName,
        DateTime dateOfBirth)
    {
        FirstName = firstName;
        LastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }

    public void SetPassword(string newPassword)
    {
        password = newPassword;
    }
}
```

Because of its automated nature, deserialization of objects which were serialized from

a different version of the object type (newer or older), can pose a problem. There are [tools available](#) to better handle these scenarios:

- The **OptionalField** attribute can be used to allow deserialization when a field is missing in the data.
- Methods to be invoked before or after serialization or deserialization can be annotated with attributes (OnSerializing, OnSerialized, OnDeserializing, and OnDeserialized).

Further details about these are out of the scope of this book. If you are interested, look up the [online documentation](#).

Binary serialization is the basis of [.NET remoting](#) – a framework for rich and efficient communication between .NET processes. That's also the only scenario for which binary serialization is still recommended. With WCF as a more modern replacement for .NET remoting, the use of binary serialization should today be limited to maintenance of legacy code.

Of course, there are many other serialization formats in use today. Since there's no built-in support in the .NET framework for them, third party libraries will need to be used in these cases, e.g.:

- [Json.NET](#) for JSON serialization (commonly used in REST services)
- [Protobuf-net](#) for use with protocol buffers – Google's highly optimized serialization format.

74

Q74. What is Reflection and what makes it possible?

Reflection is a set of APIs which can be used to get information about types at run time. This is possible because assemblies with compiled .NET code also include metadata describing all the declared types in detail. The reflection APIs use this metadata to return the requested information.

For example, a type can be determined for any instance:

```
var text = "Sample text";  
  
var type = text.GetType();
```

```
Console.WriteLine(type.FullName); // System.String
```

The `FullName` property returns the fully qualified name of the type, i.e. its namespace and type name. This also works for generic types:

```
var dictionary = new Dictionary<int, string>();

var type = dictionary.GetType();

Console.WriteLine(type.FullName);
```

The response is more complex in this case:

```
System.Collections.Generic.Dictionary`2[[System.
Int32, mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089],[System.String, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]
```

The number after the grave accent symbol (') indicates the number of generic type arguments (two in this case). The types of these arguments are listed inside square brackets and separated with commas. Each type is encapsulated in its own pair of square brackets. The fully qualified type name of a type argument is followed by the identification of the assembly in which it is declared.

This detailed information can also be accessed at run time using more specific APIs:

```
Console.WriteLine(type.IsGenericType); // True
Console.WriteLine(type.GenericTypeArguments.Length); // 2
Console.WriteLine(type.GenericTypeArguments[0].FullName); //
System.Int32
Console.WriteLine(type.GenericTypeArguments[1].FullName); //
System.String
```

It's also possible to create instances of types at run time based on a type name:

```
var type = Type.GetType("System.String");
var instance = Activator.CreateInstance(type, 'a', 5);

Console.WriteLine(instance); // aaaaa
```

The arguments of the `CreateInstance` method, except for the first one, are passed on as arguments for the constructor used to create the instance.

This also works for generic types, but the code gets more complicated because the

generic type arguments must be specified as well:

```
var typeDefinition = Type.GetType("System.Collections.Generic.  
Dictionary`2");  
var argumentType1 = Type.GetType("System.Int32");  
var argumentType2 = Type.GetType("System.String");  
var genericType = typeDefinition.MakeGenericType(argumentType1,  
argumentType2);  
var instance = Activator.CreateInstance(genericType);
```

There's much more information about a type available, e.g. its base type (notice how I can also obtain information about a type without an instance using the `typeof` operator):

```
var type = typeof(List<int>);  
Console.WriteLine(type.BaseType.FullName); // System.Object
```

If the type was derived from a base type (different from the common object base type), I could use the `BaseType` property of each base type to navigate up the inheritance hierarchy.

Information about interfaces implemented by a type is also accessible:

```
var type = typeof(List<int>);  
var interfaces = type.GetInterfaces();  
  
foreach (var interfaceType in interfaces)  
{  
    Console.WriteLine(interfaceType.FullName);  
}
```

The code above will list all the interfaces implemented by the `List<int>` type:

```
System.Collections.Generic.IList`1[[System.Int32, mscorlib, Version  
= 4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089]]  
System.Collections.Generic.ICollection`1[[System.Int32,  
mscorlib, Version = 4.0.0.0, Culture = neutral, PublicKeyToken =  
b77a5c561934e089]]  
System.Collections.Generic.IEnumerable`1[[System.Int32,
```

```
mscorlib, Version = 4.0.0.0, Culture = neutral, PublicKeyToken =
b77a5c561934e089]]
System.Collections.IEnumerable
System.Collections.IList
System.Collections.ICollection
System.Collections.Generic.IReadOnlyList`1[[System.Int32,
mscorlib, Version = 4.0.0.0, Culture = neutral, PublicKeyToken =
b77a5c561934e089]]
System.Collections.Generic.IReadOnlyCollection`1[[System.Int32,
mscorlib, Version = 4.0.0.0, Culture = neutral, PublicKeyToken =
b77a5c561934e089]]
```

While all this information can be used to check whether an instance of a type can be assigned to a variable (or member) of another type, there's a more convenient method available to perform this check:

```
var type = typeof(List<int>);
var interfaceType = typeof(IList<int>);
```

```
Console.WriteLine(interfaceType.IsAssignableFrom(type)); // True
```

It might feel a bit unintuitive that a method must be invoked on the target type to get that information, but thanks to the descriptive method name (**IsAssignableFrom**), the code is still easy enough to understand. This method is useful when creating instances dynamically based on text-based configuration data. It determines whether the created instance can be cast to the target type and used from there on:

```
var pluginAssembly = Assembly.LoadFrom(pluginAssemblyPath); // path
read from config
var pluginType = pluginAssembly.GetType(pluginTypeName); // type
read from config
if (typeof(IPlugin).IsAssignableFrom(pluginType)) {
    var pluginInstance = (IPlugin)Activator.
CreateInstance(pluginType);
    // use the plugin
}
```

Type inspection doesn't stop at type level. For each type, all its members can be enumerated as well:

```
var type = typeof(string);  
var members = type.GetMembers();
```

By default, the `GetMembers` method will return all public members. However, its overload with a parameter of type `BindingFlags` can return non-public members instead:

```
var type = typeof(string);  
var members = type.GetMembers(BindingFlags.NonPublic |  
BindingFlags.Instance | BindingFlags.Static);
```

With different combinations of `BindingFlags` values, the set of returned members can be further configured. When that's not enough, the returned array of `MemberInfo` objects can be filtered by their properties, using LINQ methods for example.

Any discovered member of a type can be accessed. It doesn't need to be `public` for that.

The available operations depend on the member type. For properties, their values can be read or written to if they have a getter or a setter respectively:

```
var text = "Sample text";  
  
var type = text.GetType();  
var property = type.GetProperty("Length");
```

```
Console.WriteLine(property.GetValue(text)); // 11
```

```
// throws System.ArgumentException : Property set method not found.  
property.SetValue(text, 10);
```

Methods can be invoked:

```
var text = "Sample text";  
var type = text.GetType();  
var method = type.GetMethod("ToUpper", new Type[0]);
```

```
Console.WriteLine(method.Invoke(text, new object[0])); // SAMPLE  
TEXT
```

When calling the `GetMethod` method, I had to specify the list of parameter types in addition to the name to identify the correct overload of the method I wanted to invoke. When invoking it, I had to pass a matching array of arguments (empty in my case).

Similarly, all other member types can be accessed as well. Anything that can be done with statically compiled code, is also possible with reflection.

But it's always more complicated and less performant.

Therefore, reflection should only be used to achieve what otherwise couldn't be done. For example, serialization is often heavily dependent on reflection.

75

Q75. What are the potential dangers of using Reflection?

In addition to being less performant than statically compiled code, reflection can also have a negative effect on the stability and security of the application. The exact impact depends on the scenario in which reflection is used.

A common use case for reflection is **application extensibility with dynamically loaded plugins**. In its simplest form, the application can specify a public interface that a class must implement to provide a specific functionality that needs to be extensible:

```
public interface IPluggable
{
    string GetString();
}
```

The application could have multiple built-in implementations for such an interface. But in addition to that, it could also include functionality to find implementations of the same interface in assemblies placed in a specific directory for plugins.

The assembly trying to extend this functionality could simply implement the interface in the same way as the application would:

```
namespace AddOns
{
    public class Pluggable : IPluggable
    {
        public string GetString()
        {
            return "Dynamically loaded";
        }
    }
}
```

All the additional work for dynamically loading these plugins would need to be done by the application. It would need to scan the plugins directory for assemblies, load them and search for suitable classes inside them:

```
var pluggables = new List<Type>();
var pluggableInterface = typeof(IPluggable);

var pluginsDir = new DirectoryInfo(pluginsPath);
var files = pluginsDir.GetFiles();

foreach (var file in files)
{
    try
    {
        var assembly = Assembly.LoadFile(file.FullName);
        var types = assembly.GetTypes();
        var pluggableTypes = types.Where(type => type.IsClass && !type.IsAbstract && pluggableInterface.IsAssignableFrom(type));
        pluggables.AddRange(pluggableTypes);
    }
    catch (Exception e)
    {
        // log errors
    }
}
```

The code you just saw does exactly that. It retrieves all the types declared in each assembly and checks whether each one of them can be assigned to the plugin interface (which effectively means that it implements that interface). It also makes sure that the type is a non-abstract class meaning that an instance of it can be created.

The application could then list all the plugins it found:

```
foreach (var pluggable in pluggables)
{
    Console.WriteLine(pluggable.FullName);
}
```

Once the user selected one of the implementations, the application could create an instance of it (using its parameterless constructor) and use it:

```
var instance = (IPluggable)Activator.
CreateInstance(selectedPluggable);
Console.WriteLine(instance.GetString());
```

The nice thing about this implementation is that the plugin class can be used in a statically bound manner from the point where it was instantiated and cast to the target interface.

Still, dynamically loading code from random assemblies can negatively affect reliability.

Of course, some of that can be avoided with a better implementation (e.g. I should have checked that there is a parameterless constructor available before using it to create the instance). But even then, the dynamically loaded code could have bugs causing the application to crash. Or it could be malicious and could try to gain access to users' files or try to damage them.

Potential risks should always be thoroughly analyzed before adding such functionality to an application.

Another possible use case for reflection is the ability to **extend functionality of a type or library by accessing and manipulating its internal members**. Of course, this only makes sense when someone else created the code and you can't simply implement the functionality in the library or make the required members public.

For example, in [Entity Framework Core](#), there's [no built-in method available for obtaining the SQL query](#) which is going to be sent to the database server. Since the query is obviously generated by EF Core, it must contain the code that generates the query. It is just not accessible through the library's public interface.

With the project being open source, it's not too difficult to find the code responsible for query generation. With the use of reflection, this code could then be invoked even if it's not publicly exposed. Smit Patel, one of Microsoft's developers, even published such code in [one issue on GitHub](#):

```
public static class IQueryableExtensions
{
    private static readonly TypeInfo QueryCompilerTypeInfo =
        typeof(QueryCompiler).GetTypeInfo();

    private static readonly FieldInfo QueryCompilerField =
        typeof(EntityQueryProvider).GetTypeInfo().DeclaredFields.First(x
        => x.Name == "_queryCompiler");

    private static readonly PropertyInfo NodeTypeProviderField =
        QueryCompilerTypeInfo.DeclaredProperties.Single(x => x.Name ==
        "NodeTypeProvider");

    private static readonly MethodInfo CreateQueryParserMethod
        = QueryCompilerTypeInfo.DeclaredMethods.First(x => x.Name ==
        "CreateQueryParser");

    private static readonly FieldInfo DataBaseField =
        QueryCompilerTypeInfo.DeclaredFields.Single(x => x.Name == "_
        database");

    private static readonly PropertyInfo DatabaseDependenciesField
        = typeof(Database).GetTypeInfo().DeclaredProperties.Single(x =>
        x.Name == "Dependencies");

    public static string ToSql<TEntity>(this IQueryable<TEntity>
        query) where TEntity : class
    {
        if (!(query is EntityQueryable<TEntity>) && !(query is
```

```
InternalDbSet<TEntity>))
{
    throw new ArgumentException("Invalid query");
}

var queryCompiler = (IQueryCompiler)QueryCompilerField.
GetValue(query.Provider);
var nodeTypeProvider = (INodeTypeProvider)NodeTypeProviderField.
GetValue(queryCompiler);
var parser = (IQueryParser)CreateQueryParserMethod.
Invoke(queryCompiler, new object[] { nodeTypeProvider });
var queryModel = parser.GetParsedQuery(query.Expression);
var database = DataBaseField.GetValue(queryCompiler);
var queryCompilationContextFactory = ((DatabaseDependencies)
DatabaseDependenciesField.GetValue(database)).
QueryCompilationContextFactory;
var queryCompilationContext = queryCompilationContextFactory.
Create(false);
var modelVisitor = (RelationalQueryModelVisitor)
queryCompilationContext.CreateQueryModelVisitor();
modelVisitor.CreateQueryExecutor<TEntity>(queryModel);
var sql = modelVisitor.Queries.First().ToString();

return sql;
}
}
```

The code above worked flawlessly with the version 2.0.0 of EF Core. However, it didn't work anymore with the next minor release, i.e. version 2.1.0.

That's the danger of using reflection in such a way.

Because the code is accessing internal members, there's no guarantee that these won't change in a future version. Private members are usually private for a reason. They give developers the freedom to refactor the code without having to keep them intact as they should with public members.

Because private members are accessed with the help of literal strings passed to the reflection APIs, the compiler also can't verify whether these members are still there. The code above will compile just fine even with the latest version of EF Core. It will

fail only at runtime when it won't find the missing private members which it wants to access.

With unit tests, the problem could at least be detected at build time when the tests would run and fail to access the requested members. But it would still require additional unplanned changes to the code in order to upgrade it to use the latest version of EF Core.

Unless such use of reflection achieves a critical functionality for an application, it should always be avoided as far as possible.

76

Q76. When to create Custom Attributes and how to inspect them at runtime?

Attributes can be used to declaratively annotate a language construct (assembly, type or its member) with some additional information which can then be used by a different piece of code.

Many attributes are already a part of the Base Class Library (BCL), e.g. the ones used by different types of serialization. In addition to those, you're also allowed to create your own attributes which you can inspect at runtime using reflection.

For example, you can create an attribute for marking type properties as not nullable:

```
[AttributeUsage(AttributeTargets.Property)]
public class NotNullableAttribute : Attribute {
    public string ErrorMessage { get; set; } = "Null value is not
allowed.";

    public NotNullableAttribute()
    { }
}
```

```

public NotNullableAttribute(string errorMessage)
{
    ErrorMessage = errorMessage;
}
}

```

To act as an attribute, the class must derive from the `System.Attribute` class. To restrict this attribute to be allowed only on properties, I annotated it with the `AttributeUsage` attribute accordingly. I also added the option to modify the default validation error message.

By applying the attribute to properties of a type you can specify which properties aren't allowed to have a null value when the instance is validated:

```

public class Person
{
    [NotNullable]
    public string FirstName { get; set; }
    [NotNullable("LastName is required.")]
    public string LastName { get; set; }
    public string MiddleName { get; set; }
}

```

For the `LastName` property, I specified a custom error message as a parameter which will map to the constructor parameter of the attribute class.

The key part is the code inspecting the attributes at runtime and performing the validation:

```

public static class ValidationExtensions
{
    public static Dictionary<string, string> GetValidationErrors(this
    object instance)
    {
        var errors = new Dictionary<string, string>();

        var attributeType = typeof(NullableAttribute);
        var instanceType = instance.GetType();
        var properties = instanceType.GetProperties();
    }
}

```

```
foreach (var property in properties)
{
    var attribute = (NotNullableAttribute)Attribute.
        GetCustomAttribute(property, attributeType);
    if (attribute != null)
    {
        var propertyValue = property.GetValue(instance);
        if (propertyValue == null)
        {
            errors.Add(property.Name, attribute.ErrorMessage);
        }
    }
}

return errors;
}
```

The validation method returns a dictionary of properties with validation errors and the corresponding error messages. I first use reflection to get the type of the instance being validated, and its properties. For every property, I then attempt to get the instance of my custom **NotNullable** attribute. If the property is annotated with this attribute, I check its value. If it's **null**, I read the error message (default or custom) from the attribute and add the error to the dictionary.

I implemented the method as an extension method which makes it convenient to use:

```
var errors = person.GetValidationErrors();
```

To achieve the same functionality without custom attributes, I would have to add validation code to each class that needs to be validated. It would be a good idea to create an interface for that:

```
public interface IValidatable
{
    Dictionary<string, string> GetValidationErrors();
}
```

Any types requiring validation would have to implement this interface:


```

public class ValidatablePerson : IValidatable
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string MiddleName { get; set; }

    public Dictionary<string, string> GetValidationErrors()
    {
        var errors = new Dictionary<string, string>();

        if (FirstName == null)
        {
            errors.Add(nameof(FirstName), "Null value is not
allowed.");
        }
        if (LastName == null)
        {
            errors.Add(nameof(LastName), "LastName is required.");
        }

        return errors;
    }
}

```

Instead of just declaratively marking the relevant properties, I must now implement imperative code performing the validation and creating the dictionary with errors.

The final code invoking the imperative implementation of the validation will look identical to the one invoking the declarative attribute-based validation:

```
var errors = person.GetValidationErrors();
```

Each approach has its own advantages and disadvantages.

For a set of predefined validation rules, the attribute-based approach is simpler to apply and maintain, at the level of types to be validated. There is some up-front work on preparing the attributes and the validation code for each rule. But in the end, there should be less code and less repeated code.

Because I must use reflection to inspect attributes, the performance won't be as good as with the second approach. Also, a dedicated `GetValidationErrors` method in each type that supports validation means more flexibility and makes it easier to implement complete custom validation logic for a specific type.

Which approach is better will depend on the individual scenario.

SECTION IX

C# 6 AND 7

77

Q77. How has C# changed since its first version?

Although it's been over 15 years since the original release of C#, the language doesn't feel that old.

The reason being it has been regularly updated. Every two or three years, a new C# version was released with additional features. Since the release of C# 7.0 in the beginning of 2017, the cadence has further increased with minor language versions. Within a year's time, three new minor language versions (7.1, 7.2, 7.3) were released.

If we were to look at the code written for C# 1.0 in 2002, it would look much different from the code that we write today. Most of the differences result from using language constructs which didn't exist back then. However, along with language development, new classes were also added to the .NET framework which take advantage of the new language features. All of this makes the C# of today much more expressive and terser.

Let's take a trip into history with an overview of major language versions. For each one of them, we will take a closer look at the most important changes and compare the code that could be written after its release, to the one that had to be written earlier. By

the time we reach C# 1.0, we will hardly be able to recognize the code as C#.

At the time of writing, the latest major language version is 7.0. With its release in 2017, it's still fairly recent, therefore its new features aren't yet used often.

*Most of us are still very much used to writing C# code
without the advantages it brings.*

The main theme of C# 7.0 was pattern matching which added support for checking types in switch statements:

```
switch (weapon)
{
    case Sword sword when sword.Durability > 0:
        enemy.Health -= sword.Damage;
        sword.Durability--;
        break;
    case Bow bow when bow.Arrows > 0:
        enemy.Health -= bow.Damage;
        bow.Arrows--;
        break;
}
```

There are multiple new language features used in the above compact piece of code:

- The **case** statements check for the type of the value in the weapon variable.
- In the same statement, a new variable of the matching type is declared which can be used in the corresponding block of code.
- The last part of the statement after the **when** keyword specifies an additional condition to further restrict when its block of code will be executed.

Additionally, the **is** operator was extended with pattern matching support, so that it can now be used to declare a new variable similarly to **case** statements:

```
if (weapon is Sword sword)
{
    // from here on, the sword variable is in scope
}
```

In earlier versions of the language without all these features, the equivalent block of code would be much longer.

```
if (weapon is Sword)
{
    var sword = weapon as Sword;
    if (sword.Durability > 0)
    {
        enemy.Health -= sword.Damage;
        sword.Durability--;
    }
}
else if (weapon is Bow)
{
    var bow = weapon as Bow;
    if (bow.Arrows > 0)
    {
        enemy.Health -= bow.Damage;
        bow.Arrows--;
    }
}
```

Several other minor features were added in C# 7.0 as well. We will mention only two of them:

Out variables allow declaration of variables at the place where they are first used as **out** arguments of a method.

```
if (dictionary.TryGetValue(key, out var value))
{
    return value;
}
else
{
    return null;
}
```

Before this feature was added, we had to declare the value variable in advance:

```
string value;
if (dictionary.TryGetValue(key, out value))
{
    return value;
}
else
{
    return null;
}
```

Tuples can be used to group multiple values into a single value on-the-fly as needed, e.g. for return values of methods:

```
public (int weight, int count) Stocktake(IEnumerable<IWeapon>
weapons)
{
    return (weapons.Sum(weapon => weapon.Weight), weapons.Count());
}
```

Without them, we had to declare a new type to do that even if we only needed it in a single place:

```
public Inventory Stocktake(IEnumerable<IWeapon> weapons)
{
    return new Inventory
    {
        Weight = weapons.Sum(weapon => weapon.Weight),
        Count = weapons.Count()
    };
}
```

C# 6.0 was released in 2015. It coincided with the full rewrite of the compiler, codenamed [Roslyn](#). An important part of it were the compiler services which have since then become widely used in Visual Studio and other editors:

- Visual Studio 2015 and 2017 are using it for syntax highlighting, code navigation, refactoring and other code editing features.
- Many other editors, such as [Visual Studio Code](#), [Sublime Text](#), [Emacs](#) and others

provide similar functionalities with the help of [OmniSharp](#), a standalone set of tooling for C# designed to be integrated in code editors.

- Many third-party static code analyzers use the language services as their basis. These can be used inside Visual Studio, but also in the build process.

There were only a few changes to the language. They were mostly syntactic sugar, but many of them are still useful enough to be commonly used today:

The **dictionary initializer** can be used to set the initial value for a dictionary:

```
var dictionary = new Dictionary<int, string>
{
    [1] = "One",
    [2] = "Two",
    [3] = "Three",
    [4] = "Four",
    [5] = "Five"
};
```

Without it, a collection initializer had to be used instead:

```
var dictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
    { 3, "Three" },
    { 4, "Four" },
    { 5, "Five" }
};
```

The **nameof** operator returns the name of a symbol:

```
public void Method(string input) {
    if (input == null) {
        throw new ArgumentNullException(nameof(input));
    }
    // method implementation
}
```


It's great for avoiding the use of strings in code which can easily go out of sync when symbols are renamed:

```
public void Method(string input)
{
    if (input == null)
    {
        throw new ArgumentNullException("input");
    }
    // method implementation
}
```

The **null conditional operator (?.)** reduces the ceremony around checking for null values:

```
var length = input?.Length ?? 0;
```

Not only is there more code required to achieve the same without it, it's much more likely that we will forget to add such a check altogether:

```
int length;
if (input == null)
{
    length = 0;
}
else
{
    length = input.Length;
}
```

The **using static** directive allows direct invoking of static methods:

```
using static System.Math;

var sqrt = Sqrt(input);
```

Before it was introduced, it was necessary to always reference the static class containing it:

```
var sqrt = Math.Sqrt(input);
```

String interpolation simplified string formatting:

```
var output = $"Length of {input} is {input.Length} characters.";
```

It not only avoids the call to `String.Format`, but also makes the formatting pattern easier to read:

```
var output = String.Format("Length of '{0}' is {1} characters.",  
input, input.Length);
```

Not to mention that having formatting pattern arguments outside the pattern makes it more likely to list them in the wrong order.

C# 5.0 was released in 2012 and introduced a very important new language feature: `async/await` syntax for asynchronous calls. It made asynchronous programming much more accessible to everyone. The feature was accompanied by an extensive set of new asynchronous methods for input and output operations in .NET framework 4.5 which was released at the same time.

With the new syntax, asynchronous code started to look very similar to synchronous code:

```
public async Task<int> CountWords(string filename)  
{  
    using (var reader = new StreamReader(filename))  
    {  
        var text = await reader.ReadToEndAsync();  
        return text.Split(' ').Length;  
    }  
}
```

If you're familiar with the `async` and `await` keywords, you will realize that the I/O call to `ReadToEndAsync` method is non-blocking. The `await` keyword releases the thread for other work until the file read completes asynchronously. Only then, the execution continues, usually back on the same thread.

Without the `async/await` syntax, the same code would be much more difficult to write and to understand:

```
public static Task<int> CountWords(string filename)
{
    var reader = new StreamReader(filename);
    return reader.ReadToEndAsync()
        .ContinueWith(task =>
        {
            reader.Close();
            return task.Result.Split(' ').Length;
        });
}
```

Notice, how I must manually compose the task continuation using the `Task.ContinueWith` method. I also can't use the `using` statement anymore to close the stream because without the `await` keyword to pause the execution of the method, the stream could be closed before the asynchronous reading was complete.

Moreover, this code is using the `ReadToEndAsync` method which was added to the .NET framework when C# 5.0 was released. Before that, only a synchronous version of the method was available. To release the calling thread for its duration, it had to be wrapped into a `Task`:

```
public Task<int> CountWords(string filename)
{
    return Task.Run(() =>
    {
        using (var reader = new StreamReader(filename))
        {
            return reader.ReadToEnd().Split(' ').Length;
        }
    });
}
```

Although this allowed the calling thread (usually the main thread or the UI thread) to do other work for the duration of the I/O operation, another thread from the thread pool was still blocked during that time. This code only seems asynchronous but is still synchronous in its core.

To do some real asynchronous I/O, a much older and more basic API in the `FileStream` class needs to be used:

```
public Task<int> CountWords(string filename)
{
    var fileInfo = new FileInfo(filename);
    var stream = new FileStream(filename, FileMode.Open);
    var buffer = new byte[fileInfo.Length];
    return Task.Factory.FromAsync(stream.BeginRead, stream.EndRead,
        buffer, 0, buffer.Length, null)
        .ContinueWith(_ =>
        {
            stream.Close();
            return Encoding.UTF8.GetString(buffer).Split(' ').Length;
        });
}
```

There was only a single asynchronous method available for reading files and it only allowed us to read the file contents as an array of bytes, therefore we are decoding the text ourselves.

Also, the above code reads the whole file at once which doesn't scale well for large files. And we're still using the `FromAsync` helper method which was only introduced in .NET framework 4 along with the `Task` class itself. Before that, we were stuck with directly using the asynchronous programming model pattern (APM) everywhere in our code, having to call the `BeginOperation` and `EndOperation` method pairs for each asynchronous operation.

No wonder, asynchronous I/O was rarely used before C# 5.0!

In 2010, C# 4.0 was released. It was focused on dynamic binding to make interoperability with COM and dynamic languages simpler. Since Microsoft Office and many other large applications can now be extended by using the .NET framework directly without depending on COM interoperability, we don't see much use of dynamic binding in C# code today.

Still, there was an important feature added at the same time, which became an integral part of the language and is commonly used today without giving it any special thought: **optional and named parameters**. They are a great alternative to writing many overloads of the same function:

```
public void Write(string text, bool centered = false, bool bold =
false)
{
    // output text
}
```

This single method can be called by providing it any combination of optional parameters:

```
Write("Sample text");
Write("Sample text", true);
Write("Sample text", false, true);
Write("Sample text", bold: true);
```

We had to write three different overloads before C# 4.0 to come as close to this as possible:

```
public void Write(string text, bool centered, bool bold)
{
    // output text
}

public void Write(string text, bool centered)
{
    Write(text, centered, false);
}

public void Write(string text)
{
    Write(text, false);
}
```

And even so, this code only supports the first three calls from the original example. Without the *named parameters* feature, we would have to create an additional method with a different name to support the last combination of parameters, i.e. to specify only text and bold parameters but keep the default value for the centered parameter:

```
public void WriteBold(string text, bool bold)
{
    Write(text, false, bold);
}
```

C# 3.0 from 2007 was another major milestone in the language development. The features it introduced all revolve around making LINQ (Language INtegrated Query) possible:

- Extension methods appear to be called as members of types although they are defined elsewhere.
- Lambda expressions provide shorter syntax for anonymous methods.
- Anonymous types are ad-hoc types which don't have to be defined in advance.

All of these contribute towards the LINQ method syntax we are all so used to today:

```
var minors = persons.Where(person => person.Age < 18)
    .Select(person => new { person.Name, person.Age })
    .ToList();
```

Before C# 3.0, there was no way to write such declarative code in C#. The functionality had to be coded imperatively:

```
List<NameAndAge> minors = new List<NameAndAge>();
foreach(Person person in persons)
{
    if (person.Age > 18)
    {
        minors.Add(new NameAndAge(person.Name, person.Age));
    }
}
```

Notice how I used the full type to declare the variable in the first line of code. The **var** keyword most of us are using all the time was also introduced in C# 3.0. While this code doesn't seem much longer than the LINQ version, keep in mind that we still need to define the **NameAndAge** type:

```

public class NameAndAge
{
    private string name;
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    private int age;
    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            age = value;
        }
    }

    public NameAndAge(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

```

The class code is much more verbose than we're used to because of two more features that were added in C# 3.0:

- Without **auto-implemented properties** I must manually declare the backing fields,

as well as the trivial getters and setters for each property.

- The constructor for setting the property values is required because there was no **initializer syntax** before C# 3.0.

Continuing our flashback, let's go to the year 2005 when C# 2.0 was released. Many consider this the first version of the language mature enough to be used in real projects. It introduced many features which we can't live without today, but the most important and impactful one of them was certainly support for **generics**.

None of us can imagine C# without generics. All the collections we're still using today are generic:

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
    sum += numbers[i];
}
```

Without generics, there were no strongly typed collections in the .NET framework. Instead of the code above, we were stuck with the following:

```
ArrayList numbers = new ArrayList();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
    sum += (int)numbers[i];
}
```

Although the code might seem similar, there's an important difference: **this code is not type safe**. I could easily add values of other types to the collection, not only ints. Also,

notice how I cast the value I retrieve from the collection before using it. I must do that because it is typed as **object** inside the collection.

Of course, such code is very error prone. Fortunately, there was another solution available if I wanted to have type safety. I could create my own typed collection:

```
public class IntList : CollectionBase
{
    public int this[int index]
    {
        get
        {
            return (int)List[index];
        }
        set
        {
            List[index] = value;
        }
    }

    public int Add(int value)
    {
        return List.Add(value);
    }

    public int IndexOf(int value)
    {
        return List.IndexOf(value);
    }

    public void Insert(int index, int value)
    {
        List.Insert(index, value);
    }

    public void Remove(int value)
    {
        List.Remove(value);
    }
}
```

```
public bool Contains(int value)
{
    return List.Contains(value);
}

protected override void OnValidate(Object value)
{
    if (value.GetType() != typeof(System.Int32))
    {
        throw new ArgumentException("Value must be of type Int32.",
            "value");
    }
}
}
```

The code using it would still be similar, but it would at least be type safe, just like we're used to with generics:

```
IntList numbers = new IntList();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
    sum += numbers[i];
}
```

However, the `IntList` collection can only be used for storing ints. I must implement a different strongly typed collection if I want to store values of a different type.

There are many other features we can't live without today which were not implemented before C# 2.0:

- Nullable value types,
- Iterators,
- Anonymous methods

C# has been the primary language for .NET development since version 1.0, but it has come a long way since then. Thanks to the features that were added to it version by version, it's staying up-to-date with new trends in the programming world and is still a good alternative to newer languages which have appeared in the meantime.

78

Q78. Which C# 6 features should I really start using?

Changes in C# 6.0 make the language less verbose. Most of them affect declarations of class members and building of expressions.

But which ones have the most positive impact on code maintainability and readability?

Wherever applicable, you should start using the following new features in your code base immediately:

- the `nameof` operator,
- the null-conditional operator,
- and support for `await` in catch and finally blocks.

I would strongly suggest you apply them even to your existing code.

The Base Class Library sometimes expects you to put symbol names (such as property

and parameter names) as string literals in your code, e.g. when your class implements the `INotifyPropertyChanged` interface or when throwing an `ArgumentException`:

```
public int GetFibonacciNumberBefore(int n)
{
    if (n < 0)
    {
        throw new ArgumentException("Negative value not allowed", "n");
    }

    // TODO: calculate Fibonacci number
    return n;
}
```

When you rename the `n` parameter, you will need to remember to change the string literal as well – the compiler will not warn you about it. The `nameof` operator in C# 6.0 is a perfect solution to this problem: the `nameof(n)` call will still compile into a string literal but you don't need to use any string constants at the source code level:

```
public int GetFibonacciNumberAfter(int n)
{
    if (n < 0)
    {
        throw new ArgumentException("Negative value not allowed",
            nameof(n));
    }

    // TODO: calculate Fibonacci number
    return n;
}
```

Not only will the compiler now warn you if you forget to rename the parameter usage in the `nameof` operator, but also the rename refactoring in Visual Studio will automatically rename it for you.

Whenever you want to invoke a property or a method on a class, you must be sure that the instance is not null beforehand, otherwise the dreaded `NullReferenceException` will be thrown at run time. This can result in a lot of trivial code, just to check for values not being null:

```
public int GetStringLengthBefore(string arg)
{
    return arg != null ? arg.Length : 0;
}
```

The **Null-conditional operator** can replace all such null checking code:

```
public int GetStringLengthAfter(string arg)
{
    return arg?.Length ?? 0;
}
```

The `arg?.Length` expression in the above code will evaluate to `arg.Length` when `arg` is not null; otherwise, it will evaluate to null. You can even cascade multiple calls one after another:

```
public string FirstItemAsString<T>(List<T> list) where T: class
{
    return list?.FirstOrDefault()?.ToString();
}
```

The `list?.FirstOrDefault()?.ToString()` expression will evaluate to null if `list` or `list.FirstOrDefault()` are null and will never throw an exception.

The null-conditional operator is not limited to properties and methods. It also works with delegates, but instead of invoking the delegate directly, you will need to call its `Invoke` method:

```
public event PropertyChangedEventHandler PropertyChanged;
private void NotifyPropertyChangedAfter(string propertyName)
{
    PropertyChanged?.Invoke(this, new
        PropertyChangedEventArgs(propertyName));
}
```

Implementing events like this also ensures that they are thread-safe, unlike their naïve implementation in previous versions of C#:

```
private void NotifyPropertyChangedBefore(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new
            PropertyChangedEventArgs(propertyName));
    }
}
```

In multithreaded scenarios, the above code could throw a `NullReferenceException`. This is in case if another thread removed the last remaining event handler after the first thread checked the event for being null, but before it called the delegate. To avoid this, `PropertyChanged` needs to be stored into a local variable before checking it for null. The null-conditional operator in C# 6.0 takes care of this automatically.

The `async` and `await` keywords introduced in C# 5.0 made asynchronous programming much easier. Instead of creating a callback for each asynchronous method call, all of the code can now be in a single `async` method, with the compiler creating the necessary plumbing instead of the developer.

Unfortunately, the `await` keyword was not supported in catch and finally blocks. This brought additional complexity when calling asynchronous methods from error handling blocks:

```
public async Task HandleAsyncBefore(AsyncResource resource, bool
throwException) {
    Exception exceptionToLog = null;
    try
    {
        await resource.OpenAsync(throwException);
    }
    catch (Exception exception)
    {
        exceptionToLog = exception;
    }
    if (exceptionToLog != null)
    {
        await resource.LogAsync(exceptionToLog);
    }
}
```

To await the asynchronous method, it had to be called outside the catch block. This was the only way to make sure it completed before executing any subsequent code (or exiting the calling method as in the example above).

Of course, if you wanted to rethrow the exception and keep the stack trace, it would get even more complicated.

With C# 6.0, none of this is required any more – async methods can simply be **awaited inside both catch and finally blocks**:

```
public async Task HandleAsync(AsyncResource resource, bool
throwException)
{
    try
    {
        await resource.OpenAsync(throwException);
    }
    catch (Exception exception)
    {
        await resource.LogAsync(exception);
    }
    finally
    {
        await resource.CloseAsync();
    }
}
```

The compiler will create all the necessary plumbing code itself, and you can be sure that it will work correctly.

79

Q79. How did tuple support change with C# 7?

Tuples were originally introduced in .NET framework 4 as a feature for simpler interoperability with dynamic languages which have already natively supported tuple types, such as Python .

At that time, a generic **Tuple** class was added to the base class library which could be used to represent tuple types:

```
public Tuple<int, int> Stocktake(IEnumerable<IWeapon> weapons)
{
    var weight = 0;
    var count = 0;
    foreach (var weapon in weapons)
    {
        weight += weapon.Weight;
        count++;
    }
    return new Tuple<int, int>(weight, count);
}
```

The Tuple was inconvenient to use, because there was no way to name its individual elements. They were always represented as predefined properties with fixed names Item1, Item2 etc. This made the code which used tuples overly difficult to understand:

```
var inventory = warehouse.Stocktake(weapons);
Console.WriteLine($"Inventory weight: {inventory.Item1}, item
count: {inventory.Item2}");
```

To determine what exactly Item1 and Item2 meant in the `inventory` Tuple above, you would need to look at the implementation of the `Stocktake` method.

The fact that the Tuple was implemented as a class, also affected performance. Every time a tuple needed to be returned from a method, a new Tuple instance was created on the heap. If such a method were to be called often in a tight loop, it would quickly lead to the garbage collector having to clean up all the instances which were created.

In C# 7.0, support for tuples was reimplemented from the ground up.

This time, the `ValueTuple` struct is used in place of the `Tuple` class. With it being a value type instead of a reference type, it is always allocated on the stack. Hence it can be used even in performance critical code since no garbage collection is required any more to release the allocated memory.

That's not the only advantage of the new support for tuples over the old one. Individual tuple elements can now have custom names:

```
public (int weight, int count) Stocktake(IEnumerable<IWeapon>
weapons)
{
    var weight = 0;
    var count = 0;
    foreach (var weapon in weapons)
    {
        weight += weapon.Weight;
        count++;
    }
    return (weight, count);
}
```

Of course, the elements can later also be referred to by using these names:

```
var inventory = warehouse.Stocktake(weapons);
Console.WriteLine($"Inventory weight: {inventory.weight}, item
count: {inventory.count}");
```

If you take a closer look at the `ValueTuple` struct, you'll see that it still has members named `Item1`, `Item2` etc. For performance reasons they are fields instead of properties, but you won't notice any other difference in comparison to the `Tuple` class. The individual elements can even still be referenced using these fixed field names, although there's no good reason why you would want to do this:

```
Console.WriteLine($"Inventory weight: {inventory.Item1}, item
count: {inventory.Item2}");
```

Custom field names are only syntactic sugar created by the compiler and supported through IntelliSense in Visual Studio with the help of the Roslyn compiler services. In compiled code, these names are not present anymore. The fixed field names `Item1`, `Item2` etc. are used everywhere instead.

The `TupleElementNamesAttribute` attribute is automatically added at compile time to preserve the element names after the code is compiled. This makes the element names available even when they are accessed from a different assembly, e.g. when a public method is returning a tuple. Both the compiler and the Visual Studio IntelliSense functionality look up the information from the attribute to keep the illusion of custom element names.

Tuples can also be used within a method. Since C# 7.1, the element names can be inferred from the variables used to initialize their values:

```
var weight = 0;
var count = 0;
var inventory = (weight, count);
foreach (var weapon in weapons)
{
    inventory.weight += weapon.Weight;
    inventory.count++;
}
```

This is not possible when a literal value is used instead of a variable. In this case, a custom name can be specified using the following syntax for creating the tuple:

```
var inventory = (weight: 0, count: 0);
```

The same syntax can also be used to change the inferred element name in C# 7.1 or to name the elements in C# 7.0 where there's no support for inferring the names.

All of this makes tuples a great replacement for anonymous types which were added to the language to avoid creating types when they are only used to group multiple values together.

Tuples even solve the limitation of anonymous types which are only useful inside a single method because they can't be declared as a return type for methods. Tuples can be returned from methods and their fields can be accessed using their custom names, even from other assemblies. They are now a better choice for returning a subset of type properties from LINQ expressions:

```
public IEnumerable<(string name, string surname)>
GetNames(IEnumerable<Person> persons)
{
    return from p in persons
           select (p.FirstName, p.LastName);
}
```

However, tuples are not supported in expression trees, therefore they can't be used with Entity Framework or in other scenarios where IQueryable<T> LINQ extension methods must be used.

To allow simpler extraction of individual values from tuples, a special deconstruction syntax has been added to the language:

```
var (weight, count) = warehouse.Stocktake(weapons);
Console.WriteLine($"Inventory weight: {weight}, item count:
{count}");
```

In the example above, the tuple returned by the `Stocktake` method is not assigned to a single variable of a tuple type. Instead, it is assigned to multiple variables grouped in a tuple-like syntax. The number and type of variables must match the structure of the returned tuple. Names don't need to match. By using the implicit `var` type, the type for each variable does not need to be specified. The correct type will be inferred automatically.

The same syntax can also be used for deconstructing any type. For this to work, that type must define one or more **Deconstruct** methods, each with any number of **out** parameters which will be mapped to variables in the target tuple:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string name, out string surname)
    {
        name = FirstName;
        surname = LastName;
    }
}
```

To deconstruct an instance of the **Person** class using its **Deconstruct** method, the same deconstruct syntax can be used as for tuples. The number and types of the variables on the left side of the assignment must match the **out** parameters of the **Deconstruct** method:

```
var person = new Person() { FirstName = "John", LastName = "Doe" };
var (name, surname) = person;
```

The **Deconstruct** method can also be implemented as an extension method which is useful if you don't have access to the class code and thus can't modify it:

```
public static void Deconstruct(this Person person, out string name,
out string surname)
{
    name = person.FirstName;
    surname = person.LastName;
}
```

Again, the deconstruction syntax is only syntactic sugar. In compiled code, the **Deconstruct** instance or extension method is called with regular **out** parameters.

80

Q80. What are Local functions and when can they be useful?

Local functions are functions which are nested inside the body of another type member, such as a method, a constructor, a property getter or setter, etc.:

```
public static void MethodWithLocalFunction()
{
    Console.WriteLine("Inside Method");
    LocalFunction();

    void LocalFunction()
    {
        Console.WriteLine("Inside LocalFunction");
    }
}
```

They are only accessible from within the member in which they are declared. They can access:

- the local variables of that member which are declared before themselves
- all the parameters of that member.

They can also be nested inside other local functions:

```
public static void MethodWithNestedLocalFunction() {
    Console.WriteLine("Inside Method");
    LocalFunction();

    void LocalFunction()
    {
        Console.WriteLine("Inside LocalFunction");
        NestedLocalFunction();

        void NestedLocalFunction()
        {
            Console.WriteLine("Inside NestedLocalFunction");
        }
    }
}
```

In the above sample, the `NestedLocalFunction` is accessible only from the parent `LocalFunction`, but not from the parent type member (`MethodWithNestedLocalFunction`). However, it still has access to all local variables of both the parent local function and its parent method (i.e. `LocalFunction` and `MethodWithNestedLocalFunction`).

In most cases, lambda expressions can be used instead of local functions to achieve the same functionality:

```
public static void MethodWithLambda() {
    Action lambda = () =>
    {
        Console.WriteLine("Inside Lambda");
    };

    Console.WriteLine("Inside Method");
    lambda();
}
```

There are some differences, though.

A lambda expression must be declared *before* it is invoked, while local functions can be called from anywhere in their scope and can therefore be conveniently placed at the end of the parent member, not interweaved with the rest of the code. This also makes it easier to call local functions recursively.

On the other hand, lambda expressions are assigned to variables, which makes it possible to change them while the code is running, e.g. to inject the implementation from outside the scope where they are declared:

```
public static IEnumerable<T> Filter<T>(IEnumerable<T> items,
Func<T, bool> predicate)
{
    return items.Where(predicate);
}
```

Also, lambda expressions don't support the iterator syntax, but local functions do. This makes them very useful in scenarios where some validation of input parameters is required at the start of an iterator:

```
public static IEnumerable<int> FibonacciSequence(int maxN)
{
    if (maxN < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(maxN));
    }
    if (maxN >= 0)
    {
        yield return 0;
    }
    if (maxN >= 1)
    {
        yield return 1;
    }
    if (maxN >= 2)
    {
        var beforePrevious = 0;
        var previous = 1;
        for (var i = 2; i <= maxN; i++)
```



```

{
    var current = beforePrevious + previous;

    beforePrevious = previous;
    previous = current;

    yield return current;
}
}
}

```

Although the code above does the job, the validation will not happen until we start iterating through the result:

```

var sequence = FibonacciSequence(-1);
// no exception thrown yet
foreach (var n in sequence)
{
    // use the numbers
}

```

If we want the exception to be thrown beforehand, we need to refactor the `FibonacciSequence` method:

```

public static IEnumerable<int>
FibonacciSequenceImmediateValidation(int maxN)
{
    if (maxN < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(maxN));
    }
    return FibonacciSequenceNoValidation(maxN);
}

private static IEnumerable<int> FibonacciSequenceNoValidation(int
maxN)
{
    if (maxN >= 0)
    {

```

```
        yield return 0;
    }
    if (maxN >= 1)
    {
        yield return 1;
    }
    if (maxN >= 2)
    {
        var beforePrevious = 0;
        var previous = 1;
        for (var i = 2; i <= maxN; i++)
        {
            var current = beforePrevious + previous;

            beforePrevious = previous;
            previous = current;

            yield return current;
        }
    }
}
```

Now, the method implementing the validation does not use the **yield** keyword. It only passes on the `IEnumerable<int>` returned by a second method which is the actual iterator. The compiler therefore won't convert the first method into a state machine and its code will execute immediately when called. (You can read more about iterators in Chapter 44: How to implement a method returning an `IEnumerable`.)

```
var sequence = FibonacciSequenceImmediateValidation(-1);
// exception thrown in the previous line
foreach (var n in sequence)
{
    // use the numbers
}
```

The `FibonacciSequenceNoValidation` method is only meant to be called from the `FibonacciSequenceImmediateValidation` method. However, it is a **private** method and can therefore be called from any other member of the same type. We can prevent that by implementing it as a local function inside `FibonacciSequenceImmediateValidation`:

```

public static IEnumerable<int>
FibonacciSequenceWithLocalFunction(int maxN)
{
    if (maxN < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(maxN));
    }
    return FibonacciSequenceNoValidation();

    IEnumerable<int> FibonacciSequenceNoValidation()
    {
        if (maxN >= 0)
        {
            yield return 0;
        }
        if (maxN >= 1)
        {
            yield return 1;
        }
        if (maxN >= 2)
        {
            var beforePrevious = 0;
            var previous = 1;
            for (var i = 2; i <= maxN; i++)
            {
                var current = beforePrevious + previous;
                beforePrevious = previous;
                previous = current;
                yield return current;
            }
        }
    }
}

```

The `FibonacciSequenceNoValidation` local function can now be called only from the body of the `FibonacciSequenceWithLocalFunction` method.

Apart from preventing unwanted calls from elsewhere, it also makes this intention clear to anyone who is reading the code. And as a bonus, there's no need to pass the

`maxN` parameter from the method to the local function as it can be directly accessed from the local function. A lambda expression couldn't be used instead because it doesn't support the iterator syntax.

There's a similar use case for local functions when using `async` and `await`:

```
public static Task AsyncMethod(int input)
{
    // perform validation
    return AsyncLocalFunction();

    async Task AsyncLocalFunction()
    {
        // actual async code
    }
}
```

When the input parameter validation is done outside the function which uses `async` and `await`, the exception can be thrown before the state machine for the `async` method is initialized. (You can learn about `async` methods in Chapter 66: In what order is code executed when using `async` and `await`.) This makes the code more performant and simplifies exception handling since the thrown exception won't be wrapped into an `AggregateException` exception as is the case with exceptions from `async` methods. (Exception handling in `async` methods is discussed in more detail in Chapter 69: How are exceptions handled in asynchronous code.)

81

Q81. What is pattern matching and how is it supported in C# 7?

Pattern matching combines two operations:

- checking whether a value matches a certain shape, and
- retrieving information from matching values.

Before C# 7, the two operations always had to be performed separately.

For example, the following code first tests if a value is of a certain type and then casts it accordingly to make all the information inside it, accessible:

```
if (weapon is Sword)
{
    var sword = weapon as Sword;
    if (sword.Durability > 0)
    {
        enemy.Health -= sword.Damage;
        sword.Durability--;
    }
}
```

```
    }  
}  
else if (weapon is Bow)  
{  
    var bow = weapon as Bow;  
    if (bow.Arrows > 0)  
    {  
        enemy.Health -= bow.Damage;  
        bow.Arrows--;  
    }  
}
```

With the introduction of pattern matching in C# 7, the **is** operator has been extended. It can now be used to declare a variable of the matching type in the same expression:

```
if (weapon is Sword sword)  
{  
    if (sword.Durability > 0)  
    {  
        enemy.Health -= sword.Damage;  
        sword.Durability--;  
    }  
}  
else if (weapon is Bow bow)  
{  
    if (bow.Arrows > 0)  
    {  
        enemy.Health -= bow.Damage;  
        bow.Arrows--;  
    }  
}
```

The newly declared variable will be in scope from the line it is declared in, until the end of the code block containing the **if** statement. However, it will only be initialized inside the inner code block of the **if** statement when the expression evaluates to true. This is required for the value to be casted to the more specific type.

```
if (!(weapon is Bow bow)) {
    // bow variable is not initialized
}
```

The resulting code is shorter and will not build if we try to access the variable where it is not initialized.

The **switch** statement has also been extended to allow similar, but even more compact code:

```
switch (weapon)
{
    case Sword sword when sword.Durability > 0:
        enemy.Health -= sword.Damage;
        sword.Durability--;
        break;
    case Bow bow when bow.Arrows > 0:
        enemy.Health -= bow.Damage;
        bow.Arrows--;
        break;
}
```

Before these changes, the **switch** statement could only be used with string or integral types, and the case statement only supported constant values.

Now, the switch statement allows values of any type because the **case** statement can perform type checks similar to the extended **is** expression. The code block of a case statement will execute when the value is of the requested type. Inside it, the declared variable will be in scope and initialized.

Optionally, a case statement can include the **when** keyword, followed by a condition. In this case, the code block of the case statement will only execute when the condition also evaluates to true.

Since case statements are not necessarily mutually exclusive anymore, they are now evaluated in the order as they appear in the code and only the first one which matches, will have its code block executed.

The only exception to this rule is the default code block which will only execute if no case statement is matched. The behavior will be the same even if the default block is

listed as the first one in the switch statement:

```
switch (weapon) {
    default:
        throw new ArgumentException($"Unsupported weapon type {weapon.
            GetType()}", nameof(weapon));
    case Sword sword when sword.Durability > 0:
        enemy.Health -= sword.Damage;
        sword.Durability--;
    break;
    case Bow bow when bow.Arrows > 0:
        enemy.Health -= bow.Damage;
        bow.Arrows--;
    break;
}
```

In the above sample, I introduced a bug which might not be completely obvious.

The case statements checking the type of the value will only match if the value is not null. There's no such precondition for the default code block. It will execute whenever none of the other blocks match, even if the value of the variable tested is `null`.

To prevent a `NullReferenceException` exception from being thrown, another newly introduced type of `case` statement can be used:

```
switch (weapon) {
    default:
        throw new ArgumentException($"Unsupported weapon type {weapon.
            GetType()}", nameof(weapon));
    case Sword sword when sword.Durability > 0:
        enemy.Health -= sword.Damage;
        sword.Durability--;
    break;
    case Bow bow when bow.Arrows > 0:
        enemy.Health -= bow.Damage;
        bow.Arrows--;
    break;
    case null:
    break;
}
```


Now, the `case null` block will match the null value, quietly ignoring it. The default block will only execute for non-null values which are not yet covered by any of the other cases. It will throw an exception which will remind us that we still need to handle that type correctly.

Pattern matching is commonly used in functional programming. In object-oriented languages, the same result can usually be achieved using polymorphism.

To implement the above sample with polymorphism, a common interface with a single method could be defined for all weapons. The code from individual cases of the switch statement would be moved to the implementation of that method for each type. A type wouldn't compile if it didn't implement that method.

Which approach is more appropriate depends on the individual scenario and the coding practices used.

82

Q82. What is a Discard and when can it be used?

When deconstructing a tuple, it might happen that you are not interested in the values of all its elements:

```
var (weight, count) = warehouse.Stocktake(weapons);  
Console.WriteLine($"Inventory weight: {weight}");
```

In the above example, we deconstructed the tuple returned by the `Stocktake` method into `weight` and `count` variables, but we only used the `weight` variable. Instead of the `count` variable, we could use a discard:

```
var (weight, _) = warehouse.Stocktake(weapons);  
Console.WriteLine($"Inventory weight: {weight}");
```

The `_` character in the code above is called a discard. Although it looks like a variable name, it isn't. If we try to reference it later, the code won't compile:

```
Console.WriteLine(_);
```

The following compiler error will be emitted for the line of code shown above:

"The name '_' does not exist in the current context."

So, when using a discard, you will not only express your intent of *not* using the value, the compiler will also prevent you from using it.

Discards can of course also be used when you're deconstructing an instance of a type with a matching **Deconstruct** method:

```
var person = new Person() { FirstName = "John", LastName = "Doe",
Age = 42 };
var (name, _, _) = person;
Console.WriteLine($"{name}");
```

As you can see in this example, I can use the discard in multiple places inside a single expression, even for values of different types. If it were a real variable, the compiler wouldn't allow that.

Use of discards is not limited to deconstruction syntax. They can also be very useful in pattern matching:

```
switch (weapon)
{
    case Bow bow:
        Console.WriteLine($"It's a bow with {bow.Arrows} arrows
        left.");
        break;
    case Sword _:
        Console.WriteLine("It's a sword.");
        break;
    default:
        Console.WriteLine("It's an unrecognized weapon.");
        break;
}
```

The **case** statement for the type pattern requires both a type and a variable name to be specified. I've done this in the first case statement above, where I later needed to access the Bow-typed **bow** variable to read one of its properties. In the second case statement, I don't need to read any value from the Sword-typed variable, therefore

I used a discard instead of a variable name. Just like in the deconstruction example, there is no variable named `_` available in the corresponding code block.

Another common use case for discards are methods with `out` parameters. Instead of passing a real variable when calling such a method, a discard can be used:

```
var isNumber = Int32.TryParse(input, out _);
```

Since I'm only interested in whether the input string is a valid integer numeric value and not in the value itself, I'm ignoring the value of the output parameter by using a discard.

However, discard cannot always be used in place of a variable. Using `_` in such cases is still valid C# code, but it will be used as the name for the variable:

```
var validNumbers = numbers.Where(_ => _ > 3).ToArray();
```

Since in lambda expressions discards can't be used in place of parameter declarations, the `_` in the above code is a real variable. Therefore, we can use it in a conditional expression.

If `_` is already a valid variable in a scope, it won't behave as a discard even in places where discards are allowed. This behavior is required to ensure backward compatibility for code where `_` was already used as a variable before discards existed.

Here's a contrived example demonstrating this scenario:

```
var numbers = new[] { "4", "3", "Two", "1", "0" };
var validNumbers = numbers.Where((number, _) =>
{
    if (Int32.TryParse(number, out _))
    {
        Console.Write(_);
        return true;
    }
    return false;
}).ToArray();
```

Since the `_` variable is declared in the parameter list of the lambda passed to the `Where` method, it is still in scope when calling the `TryParse` method. Therefore, the

above code will output "4310", i.e. the successfully parsed numbers concatenated together.

If the `_` in the `TryParse` method call was treated as a discard, it would not modify the value in the `_` variable and the output would be "0134", i.e. the indices of the numbers which can be parsed. This means that the behavior of the code would change when compiled with a new compiler with support for discards. To avoid this undesired side effect, discards are not supported when a variable named `_` is in scope.

83

Q83. What improvements for Asynchronous programming have been introduced in C# 7?

The `async` and `await` functionality was first introduced in C# 5. It was further improved in C# 6 when support was added for using the `await` keyword in `catch` and `finally` code blocks.

Before C# 7.0, the methods using `async` and `await` either had no return value (and couldn't be awaited) or were required to return the `Task` or `Task<T>` type:

```
public async Task<int> ReturnsTask()
{
    await Task.Delay(10);
    return 42;
}
```

With C# 7.0, other types can be used as well.

The only requirement is that they have a `GetAwaiter` method which is used by the compiler to generate the plumbing code necessary for `async` and `await` to work. The

main reason behind this change was to address the performance sensitive scenarios which suffer because `Task` and `Task<T>` are reference types and need to be allocated on the heap for each asynchronous call.

Along with support in the language, `ValueTask<T>` was made available in a standalone `System.Threading.Tasks.Extensions` NuGet package which can be used in place of `Task<T>`:

```
public async ValueTask<int> ReturnsValueTask()
{
    await Task.Delay(10);
    return 42;
}
```

Since `ValueTask<T>` is a `struct` and not a `class`, it doesn't require any memory allocation on the heap. This can prove beneficial especially when asynchronous methods are being called in tight loops. It has its own disadvantages though, and therefore shouldn't be blindly used in place of `Task<T>` everywhere:

- It is larger than `Task<T>` with two fields instead of a single one.
- It can't be used directly with existing methods expecting a `Task` or `Task<T>`, such as `Task.WhenAll` and `Task.WhenAny`.

The official recommendation is to only use `TaskValue<T>` when during performance analysis, the use of `Task<T>` is identified as a bottleneck. Even then, the results of replacing `Task<T>` with `ValueTask<T>` should be measured to confirm the benefits.

Since C# 7.1, four additional signatures are supported for the `Main` method as the program entry point:

```
static Task Main();
static Task<int> Main();
static Task Main(string[] args);
static Task<int> Main(string[] args);
```

Any of these can now be used in addition to the signatures which were already supported earlier:

```
static void Main();
```

```
static int Main();  
static void Main(string[] args);  
static int Main(string[] args);
```

This change simplifies the use of asynchronous methods in a console application:

```
static async Task Main(string[] args)  
{  
    Console.WriteLine("Program start");  
    for (int i = 0; i < 10; i++)  
    {  
        await Task.Delay(1000);  
        Console.WriteLine($"{i + 1} seconds passed.");  
    }  
}
```

Because asynchronous methods can only be awaited from inside other asynchronous methods, previously, this could only be achieved by using the following boilerplate code:

```
static void Main(string[] args)  
{  
    MainAsync(args).GetAwaiter().GetResult();  
}  
  
static async Task MainAsync(string[] args)  
{  
    // asynchronous code  
}
```

Now, the required boilerplate code will be automatically generated by the compiler.

84

Q84. How was support for passing values by reference expanded in C# 7?

For a long time, C# only supported passing values by reference for method parameters. You can read more about that in Chapter 18 – "How can method parameters be passed by reference?".

In C# 7.0, support for **return values and local variables by reference** was introduced. They can be used instead of regular return values and local variables to prevent unnecessary copying of data.

However, this also affects the behavior.

Since a local variable by reference is pointing at the original memory location, any changes to the value at that location will of course also affect the local variable value:

```
[Test]
public void LocalVariableByReference()
{
    var terrain = Terrain.Get();
```

```
// assign a return value by reference to a local variable by
reference
ref TerrainType terrainType = ref terrain.GetAt(4, 2);
// check the local variable
Assert.AreEqual(TerrainType.Grass, terrainType);

// modify enum value at the original location
terrain.BurnAt(4, 2);
// local value was also affected
Assert.AreEqual(TerrainType.Dirt, terrainType);
}
```

In the above example, `terrainType` is a local variable by reference, and `GetAt` is a method returning a value by reference:

```
public ref TerrainType GetAt(int x, int y) => ref terrain[x, y];
```

The returned value is a reference to the original memory location. When assigned to a local variable by reference, that variable also points at the same memory location.

Before C# 7.3, there was no way to conditionally bind such a variable by reference to a different expression, similar to what the ternary operator (also known as the conditional operator) does when binding by value:

```
var max = a > b ? a : b;
```

Since a variable bound by reference cannot be reassigned to refer to a different memory location, this limitation could not be worked around with an `if` statement:

```
var a = 1;
var b = 2;

ref var max = ref b; // requires initialization
if (a > b)
{
    max = ref a;      // not allowed in C# 7.2
}
```

For some cases, the following method can work as a replacement:

```

ref T BindConditionally<T>(bool condition, ref T trueExpression,
ref T falseExpression)
{
    if (condition)
    {
        return ref trueExpression;
    }
    else
    {
        return ref falseExpression;
    }
}

// method call
ref var max = ref BindConditionally(a > b, ref a, ref b);

```

It will however fail if one of the arguments cannot be evaluated when the method is called:

```

ref var firstItem = ref BindConditionally(emptyArray.Length > 0, ref
emptyArray[0], ref nonEmptyArray[0]);

```

This will throw an `IndexOutOfRangeException` because `emptyArray[0]` will still be evaluated.

With the **conditional ref expression** that was introduced in C# 7.2, the described behavior can now be achieved. Just like with the existing conditional operator, only the selected alternative will be evaluated:

```

ref var firstItem = ref (emptyArray.Length > 0 ? ref emptyArray[0] :
ref nonEmptyArray[0]);

```

The ability to **reassign local variables and parameters bound by reference** to refer to a different memory location was added in C# 7.3. With this change, we can write the following code to implement the functionality equivalent to the conditional **ref** expression introduced in C# 7.2:

```
ref var max = ref b; // requires initialization
if (a > b)
{
    max = ref a;      // not allowed in C# 7.2
}
```

In performance sensitive applications, structs are often passed by reference to the called function, not because it should be able to modify the values, but to avoid copying of values. There was no way to express that in C# before version 7.2, therefore the intention could only be explained in documentation or code comments, which was purely informal and without assurance.

To address this issue, C# 7.2 includes support for **read-only parameters passed by reference**:

```
Vector3 Normalize(in Vector3 value)
{
    // returns a new unit vector from the specified vector
    // signature ensures that the input vector cannot be modified

    // value.X = 5; // will not compile
    value.Scale(2); // will be invoked on a copy to not modify value
}
```

The compiler will prevent any changes to the input parameter if it is a **struct**. Assignments to its fields and properties won't compile. For method invocations, a defensive copy will be used because the compiler can't determine whether they will modify the parameter or not.

Unlike the other two types of parameters passed by reference (**ref** and **out**), the use of the **in** keyword when invoking such a method is optional. However, without it, the compiler will prefer using the overload with parameters passed by value if it exists because it is considered a better match.

This feature also allows passing compile time constant values as read-only parameters by reference:

```
var result = Normalize(new Vector3(1, 1, 1));
```

The feature is not restricted to structs. It will also work with reference types but is most beneficial when used with structs as it can avoid unnecessary copying of values.

To further help the compiler with code optimization, another new feature can be used: **read-only structs**. They will only compile if they are immutable:

```
readonly struct ReadonlyStruct
{
    public int ImmutableProperty { get; }
    //public int MutableProperty { get; set; } // will not compile
}
```

When used as read-only parameters by reference, the compiler doesn't need to create defensive copies for invoking the methods of these structs, as it knows that they cannot modify the struct.

Read-only return values by reference are somewhat similar to read-only parameters by reference:

```
struct Vector3
{
    // other struct members omitted for brevity

    private static readonly Vector3 zero = new Vector3(0, 0, 0);
    public static ref readonly Vector3 Zero => ref zero;
}
```

They return a reference to the caller instead of a copy and the compiler doesn't allow it to be modified if it is assigned to a variable which is also declared as **ref readonly**. When the value is assigned to a variable that's not **ref readonly**, it will be copied, and this restriction will be removed:

```
ref readonly var zero = ref Vector3.Zero;
// zero.X = 1; // will not compile
var copy = zero;
copy.X = 1;
```

The final new feature is support for structs which must be allocated on the stack, i.e. **ref struct types**. This imposes several restrictions on how they can be used:

- They can't be boxed, e.g. by casting to **object** type or assigning to a dynamic-typed variable.
- They can't be members of classes or regular structs.
- They can't be used in lambda expressions or local functions.
- They can't be used in asynchronous methods.

All of this makes them safe for interop use with non-managed APIs.

The main motivation for them was the implementation of the `Span<T>` type, which provides an array-like abstraction over a part of contiguous memory, such as a larger array, or some memory on stack or even memory originating from native code. This allows processing of sub-arrays in-place, without any copying of data, making it much more efficient:

```
[Test]
public void ProcessSubArrayInPlace()
{
    var array = new[] { 1, 2, 3, 4, 5 }; // initialize array
    Span<int> span = array; // implicit cast
    var subSpan = span.Slice(start: 1, length: 3); // reference
    part of the array
    (subSpan[0], subSpan[2]) = (subSpan[2], subSpan[0]); // swap
    items

    CollectionAssert.AreEqual(new[] { 1, 4, 3, 2, 5 }, array);
}
```

To use **Span** in the .NET framework, the **System.Memory** standalone NuGet package must be installed. The **Span** type is included in .NET Core since version 2.1.

85

Q85. How to handle errors in recent versions of C#?

The idiomatic way to express error conditions in the .NET framework is by throwing exceptions. In C#, we can handle them using the try-catch-finally statement:

```
try
{
    // code which can throw exceptions
}
catch
{
    // code executed only if an exception was thrown
}
finally
{
    // code executed whether an exception was thrown or not
}
```

Whenever an exception is thrown inside the `try` block, the execution continues in the

`catch` block. The `finally` block executes after the `try` block successfully completes or when exiting the `catch` block i.e. either successfully or with an exception that might be thrown inside the catch block.

In the `catch` block, we usually need information about the exception which we are handling. To get this information, we use the following syntax:

```
catch (Exception ex)
{
    // code can access exception details in variable ex
}
```

The type used (`Exception` in our case) specifies which exceptions will be caught by the `catch` block (all in our case, as `Exception` is the base type of all exceptions). Any exceptions that are not of the given type or its descendants, will not be caught.

We can even add multiple `catch` blocks to a single `try` block. In this case, the exception will be caught by the first catch block with matching exception type:

```
catch (FileNotFoundException ex)
{
    // code will only handle FileNotFoundException and its
    // descendants
}
catch (Exception ex)
{
    // code will handle all the other exceptions
}
```

In C# 6, support for exception filters was added which can be used to further restrict when an exception will be caught by a specific catch block:

```
catch (ArgumentException ex) when (ex.ParamName == "input")
{
    // code will only handle ArgumentException for parameter named
    // "input"
}
```

All of this allows us to handle different types of exceptions in different ways. We can

recover from expected exceptions in a very specific way, for example:

- If a user selected a non-existing or invalid file, we can allow her/him to select a different file or cancel the action.
- If a network operation timed out, we can retry it or invite the user to check her/his network connectivity.

For remaining unexpected exceptions, e.g. a `NullReferenceExceptions` caused by a bug in code, we can show the user a generic error message, giving him an option to report the error, or log the error automatically without user intervention.

It is also very important to avoid swallowing exceptions:

```
catch (Exception ex)
{ }
```

Doing this is bad for both the user and the developer. The user might incorrectly assume that an action succeeded, whereas it failed silently or did not complete. The developer will not get any information about the exception, not being aware that she/he might need to fix something in the application.

Silently hiding errors is only appropriate in very specific scenarios, for example to catch exceptions thrown when attempting to log an error in the handler for unexpected exceptions. Even if we attempted to log this new error or retried logging the original error, there is a high probability that it would still fail. Silently aborting is probably the lesser evil in this case.

When writing exception-handling code, it is important to do it in the right place. You might be tempted to do it as close to the origin of the exception as possible, e.g. at the level of each individual method:

```
void MyMethod() {
    try {
        // actual method body
    }
    catch (Exception ex) {
        // exception handling code
    }
}
```

This is often appropriate for exceptions, which you can handle programmatically without any user interaction, i.e. your application can fully recover from the exception and still successfully complete the requested operation, and therefore the user does not need to know that the exception occurred at all.

However, if the requested action failed because of the exception, the user needs to know about it.

In a desktop application, it would technically be possible to show an error dialog from the same method where the exception originally occurred. But in the worst case, this could result in a cascade of error dialogs, because the methods called afterwards might also fail – e.g. they will not have the required data available:

```
void UpdatePersonEntity(PersonModel model)
{
    var person = GetPersonEntity(model.Id);
    ApplyChanges(person, model);
    Save(person);
}
```

Let us assume an unrecoverable exception is thrown inside `GetPersonEntity` (e.g., there is no `PersonEntity` with the given `Id`):

- `GetPersonEntity` will catch the exception and show an error dialog to the user.
- Because of the previous failure, `ApplyChanges` will fail to update the `PersonEntity` with the value of `null`, as returned by the first method. According to the policy of handling the exception where it happens, it will show a second error dialog to the user.
- Similarly, `Save` will also fail because of `PersonEntity` having null value, and will show the third error dialog in a row.

To avoid such situations, you should only handle the unrecoverable exceptions and show error dialogs in methods directly invoked by a user action. In a desktop application, these will typically be event handlers:

```
private void SaveButton_Click(object sender, EventArgs e)
{
    try
    {
        UpdatePersonEntity(model); // shouldn't show any error dialogs
    }
    catch (Exception ex)
    {
        // show error dialog
    }
}
```

In contrast, `UpdatePersonEntity` and any other methods called by it should not catch any exceptions they cannot properly handle. Any exception will then bubble up to the event handler, which will show the only error dialog to the user.

This also works well in a web application.

In an MVC application, for example, the only methods directly invoked by the user are action methods in controllers. In response to unhandled exceptions, these methods can redirect the user to an error page instead of the regular one.

To make the process of catching unhandled exceptions in entry methods simpler, the .NET framework provides means for global exception handling. Although the details depend on the type of the application (desktop/web), the global exception handler is always called if the exception is not handled in the outermost method, as the last chance to prevent the application from crashing.

In WPF, a global exception handler can be hooked up to the application's `DispatcherUnhandledException` event:

```
public partial class App : Application
{
    public App()
    {
        DispatcherUnhandledException += App_
        DispatcherUnhandledException;
    }
}
```

```
private void App_DispatcherUnhandledException(object sender,
DispatcherUnhandledExceptionEventArgs e)
{
    var exception = e.Exception; // get exception
    // ...                       show the error to the user
    e.Handled = true;           // prevent the application from
    crashing
    Shutdown();                 // quit the application in a
    controlled way
}
}
```

In ASP.NET, the global exception handler is convention based – a method named `Application_Error` in `global.asax`:

```
protected void Application_Error(object sender, EventArgs e)
{
    var exception = Server.GetLastError(); // get exception
    Response.Redirect("error");           // show error page to user
}
```

What exactly should a global exception handler do?

From the user standpoint, it should display a friendly error dialog or error page with instructions on how to proceed, e.g. retry the action, restart the application, contact support, etc. For the developer, it is even more important to log the exception details for further analysis:

```
File.AppendAllText(logPath, exception.ToString());
```

Calling `ToString()` on an exception will return all its details, including the description and call stack in text format. This is the minimum amount of information you want to log. To make further analysis easier, you can include additional information, such as current time and any other useful information.

SECTION X

A Look into the Future

86

Q86. What Functional programming features are supported in C#?

Functional programming is an alternative programming paradigm to the more popular and common object-oriented programming paradigm.

There are several key concepts that differentiate it from other programming paradigms. Let's start by providing definitions for the most common ones, so that we will recognize them when we see them applied throughout this chapter.

Basic building blocks of functional programs are **pure functions**. A pure function is defined by the two properties:

- Its result depends solely on the arguments passed to it. No internal or external state affects it.
- It doesn't cause any side effects. The number of times it is called will not change the program behavior.

Because of these properties, a function call can be safely replaced with its result. For example, to improve performance, we can cache the results of a computationally

intensive function for each combination of its arguments (a technique known as **memoization**).

Pure functions lend themselves well to function composition. This is a process of combining two or more functions into a new function, which returns the same result as if all its composing functions were called in sequence. If **ComposedFn** is a function composition of **Fn1** and **Fn2**, then the following assertion will always pass:

```
Assert.That(ComposedFn(x), Is.EqualTo(Fn2(Fn1(x))));
```

Composition is an important part of making functions reusable.

Having functions as arguments to other functions can further increase their reusability. Such **higher-order functions** can act as generic helpers. For example, they can apply another function passed multiple times as an argument, e.g. on all items of an array:

```
var anyMinors = Array.Exists(persons, IsMinor);
```

In the above code, **IsMinor** is a function defined elsewhere. For this to work, support for **first-class functions** is required, i.e. functions must be treated as first-class language constructs just like value literals are.

Data is always represented with **immutable objects**, i.e. objects that cannot change their state after they have been initially created. Whenever a value changes, a new object must be created instead of modifying the existing one. Because all objects are guaranteed to not change, they are inherently thread-safe, i.e. they can be safely used in multithreaded programs with no threat of race conditions.

As a direct consequence of functions being pure and objects being immutable, there is **no shared state** in functional programs. Functions can act only based on their arguments, which they cannot change and with that, affect other functions receiving the same arguments. The only way they can affect the rest of the program is through the result they return, which will be passed on as arguments to other functions. This prevents any kind of hidden cross-interaction between the functions.

Unless one function directly depends on the result of the other, they can be safely run in any order or even in parallel.

With the above basic building blocks, functional programs end up being more declarative than imperative, i.e. instead of describing how to calculate the result, the

programmer rather describes what to calculate.

The following two functions for making an array of strings lower case clearly demonstrate the difference between the two approaches:

```
string[] Imperative(string[] words)
{
    var lowerCaseWords = new string[words.Length];
    for (int i = 0; i < words.Length; i++)
    {
        lowerCaseWords[i] = words[i].ToLower();
    }
    return lowerCaseWords;
}

string[] Declarative(string[] words)
{
    return words.Select(word => word.ToLower()).ToArray();
}
```

Although you will hear about many other functional concepts, such as monads, functors, currying, referential transparency and others, these should suffice to give you the basic idea of what functional programming is and how it differs from object-oriented programming.

You can implement many of the functional programming concepts in C#.

Since the language is primarily object-oriented, the defaults don't always guide you towards writing functional code. But with intent and enough self-discipline, your code can become much more functional.

You are very likely used to writing mutable types in C#, but with very little effort, they can be made **immutable**:

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }
    public int Age { get; }
```



```
public Person(string firstName, string lastName, int age)
{
    FirstName = firstName;
    LastName = lastName;
    Age = age;
}
}
```

Properties without setters can't be assigned a different value after the object has been initially created. For the object to be truly immutable, all of the properties must also be of immutable types. Otherwise their values can be changed by mutating the properties instead of assigning a new value to them.

The `Person` type above is immutable because `string` is also an immutable type, i.e. its value cannot be changed as all its instance methods return a new string instance. However, this is an exception to the rule and most .NET framework classes are mutable. If you want your type to be immutable, you should not use any built-in types other than primitive types and strings as public properties.

To change a property of the object, e.g. to change the person's first name, a new object needs to be created:

```
public static Person Rename(Person person, string firstName)
{
    return new Person(firstName, person.LastName, person.Age);
}
```

When a type has many properties, writing such functions can become quite tedious. Therefore, it is a good practice for immutable types to implement a `With` helper method for such scenarios:

```
public Person With(string firstName = null, string lastName = null,
int? age = null)
{
    return new Person(firstName ?? this.FirstName, lastName ?? this.
LastName, age ?? this.Age);
}
```

This method creates a copy of the object with any number of properties modified.

Our **Rename** function can now simply call this helper method to create the modified person:

```
public static Person Rename(Person person, string firstName)
{
    return person.With(firstName: firstName);
}
```

The advantages might not be obvious with only two properties, but no matter how many properties the type consists of, this syntax allows us to only list the properties we want to modify as named arguments.

Making functions **pure** requires even more discipline than making objects immutable. There are no language features available to help the programmer ensure that a particular function is pure. It is your own responsibility to not use any kind of internal or external state, to not cause side effects and to not call any other functions that are not pure.

Of course, there is also nothing stopping you from only using the function arguments and calling other pure functions, thus making the function pure. The **Rename** function above is an example of a pure function: it does not call any non-pure functions or use any data other than the arguments passed to it.

Multiple functions can be **composed** into one by defining a new function, which calls all the composed functions in its body (let us ignore the fact that there is no need to ever call **Rename** multiple times in a row):

```
public static Person MultiRename(Person person)
{
    return Rename(Rename(person, "Jane"), "Jack");
}
```

The signature of the **Rename** function forces us to nest the calls, which can become difficult to read and comprehend as the number of function calls increases. If we use the **With** method instead, our intent becomes clearer:

```
public static Person MultiRename(Person person)
{
    return person.With(firstName: "Jane").With(firstName: "Jack");
}
```

To make the code even more readable, we can break the chain of calls into multiple lines, keeping it manageable, no matter how many method calls we compose into one function:

```
public static Person MultiRename(Person person)
{
    return person
        .With(firstName: "Jane")
        .With(firstName: "Jack");
}
```

When calls are nested, as in the case of calling the `Rename` function inside the `MultiRename` function, there is no good way to split the lines.

Of course, the `With` method allows the chaining syntax due to the fact that it is an instance method. However, in functional programming, functions should be declared separately from the data they act upon, like the `Rename` function is. While functional languages have a pipeline operator (`|>` in F#) to allow chaining of such functions, we can take advantage of extension methods in C# instead:

```
public static class PersonExtensions
{
    public static Person Rename(this Person person, string firstName)
    {
        return person.With(firstName: firstName);
    }
}
```

This allows us to chain non-instance method calls the same way as we can chain instance method calls:

```
public static Person MultiRename(Person person)
{
    return person.Rename("Jane").Rename("Jack");
}
```

To get a taste of functional programming in C#, you don't need to write all the objects and functions yourself. There are some readily available functional APIs in the .NET framework for you to utilize.

We have already mentioned string and primitive types as immutable types in the .NET framework. However, there is also a selection of **immutable collection types** available.

Technically, they are not really a part of the .NET framework, since they are distributed out-of-band as a stand-alone NuGet package (System.Collections.Immutable). On the other hand, they are an integral part of .NET Core, the new open-source cross-platform .NET runtime. You can read more about them in Chapter 48 – “How are immutable collections different from other collections?”.

A much better-known functional API in the .NET framework is **LINQ** (the entire section VI of the book is dedicated to it). Although it is not commonly advertised as being functional, it manifests many previously introduced functional properties.

If we take a closer look at LINQ extension methods, it quickly becomes obvious that almost all of them are declarative in nature: they allow us to specify what we want to achieve, not how:

```
var result = persons
    .Where(p => p.FirstName == "John")
    .Select(p => p.LastName)
    .OrderBy(s => s.ToLower())
    .ToList();
```

The above query returns an ordered list of last names of people named John. Instead of providing a detailed sequence of operations to perform, we only described the desired result. The available extension methods are also easy to compose using the chaining syntax.

Although LINQ functions are not necessarily acting on immutable types, they are still pure functions, unless made impure by passing mutating functions as arguments. They are implemented to act on collections of type **IEnumerable** which is a read-only interface. They don't modify the items in the collection. Their result only depends on the input arguments and they don't create any global side effects, as long as the functions passed as arguments are also pure. In the example above, neither the persons collection, nor any of the items in it will be modified.

Many LINQ functions are higher-order functions: they accept other functions as arguments. In the sample code above, lambda expressions are passed in as function arguments, but they could easily be defined elsewhere and passed in instead of created inline:

```
public bool FirstNameIsJohn(Person p)
{
    return p.FirstName == "John";
}

public string PersonLastName(Person p)
{
    return p.LastName;
}

public string StringToLower(string s)
{
    return s.ToLower();
}

var result = persons
    .Where(FirstNameIsJohn)
    .Select(PersonLastName)
    .OrderBy(StringToLower)
    .ToList();
```

When function arguments are as simple as in our case, the code will usually be easier to comprehend with inline lambda expressions instead of separate functions. However, as the implemented logic becomes more complex and reusable, having them defined as standalone functions starts to make more sense.

87

Q87. What are some major new features in C# 8?

The latest major version of C# (8.0) was released in its final form in September 2019.

At the time of release, C# 8.0 could only be used with .NET Core 3.0 and .NET Standard 2.1 projects.

In the future, support will probably be added to other runtimes compatible with .NET Standard 2.1 (i.e. Xamarin, UWP, Unity and Mono). No official support is planned for .NET framework. Although the language version can be manually set to 8.0 in a .NET framework project file, some language features won't work in this case.

Visual Studio 2019 16.3 or newer is required to create or open .NET Core 3.0 and .NET Standard 2.1 projects. In Visual Studio Code, the latest version of the C# extension is required for full language support.

C# 8.0 introduces new syntax for expressing a **range** of values:

```
Range range = 1..5;
```

The starting index of a range is inclusive, and the ending index is exclusive. Alternatively, the ending can be specified as an offset from the end:

```
Range range = 1..^1;
```

The new type can be used as indexer for arrays. Both ranges specified above will give the same result when used with the following snippet of code:

```
var array = new[] { 0, 1, 2, 3, 4, 5 };
var subArray = array[range];
```

```
Assert.AreEqual(new int[] { 1, 2, 3, 4 }, subArray);
```

Open-ended ranges are supported as well:

```
Assert.AreEqual(new int[] { 1, 2, 3, 4, 5 }, array[1..]);
Assert.AreEqual(new int[] { 0, 1, 2, 3, 4 }, array[..^1]);
```

The syntax for specifying offset from the end is not limited to ranges. It can also be used to specify an **index**, again as an offset:

- from the start:

```
Index index = 5;
```

- or from the end:

```
Index index = ^1;
```

When used as an indexer for arrays, the value at the given offset will be returned. Both the indices shown above specify the same value in the following array:

```
var array = new[] { 0, 1, 2, 3, 4, 5 };
Assert.AreEqual(5, array[index]);
```

There's no need to add additional indexers for the new Range and Index types to make existing types usable with the new syntax. The compiler implicitly adds support for the new indexers to the types which already have the following members:

- For the Index indexer, the `int` indexer and either the `Length` or the `Count` property are required. The `int` indexer can then be used instead of the missing Index indexer:

```
int offset = index.GetOffset(array.Length);
Assert.AreEqual(array[index], array[offset]);
```

- For the Range indexer, the `Slice` method and again either the `Length` or the `Count` property are required. The `Slice` method can then be used instead of the missing Range indexer:

```
(int offset, int length) = range.GetOffsetAndLength(span.
Length);
Assert.AreEqual(span[range].ToArray(), span.Slice(offset,
length).ToArray());
```

The `String` type is a special case.

It supports the new indexer syntax although it doesn't have all the required members listed above. For the Range indexer, the `Substring` method is used instead of the `Slice` method:

```
Assert.AreEqual('5', "012345"^[1]);
Assert.AreEqual("1234", "012345"[1..^1]);
```

Nullable reference types were already considered in the early stages of C# 7.0 development but were postponed until the next major version (i.e. till C# 8.0). The goal of this feature is to help developers avoid unhandled `NullReferenceException` exceptions.

The core idea is to allow variable type definitions to specify whether they can have `null` value assigned to them or not:

```
IWeapon? canBeNull;
IWeapon cantBeNull;
```

Assigning a `null` value or a potential `null` value to a non-nullable variable would result in a compiler warning (the developer could configure the build to fail in case of such warnings, to be extra safe):


```

canBeNull = null;           // no warning
cantBeNull = null;          // warning
cantBeNull = canBeNull;     // warning

```

Similarly, warnings would be generated when dereferencing a nullable variable without checking it for `null` value first:

```

canBeNull.Repair();         // warning
cantBeNull.Repair();        // no warning
if (canBeNull != null)
{
    canBeNull.Repair();     // no warning
}

```

The problem with such a change is that it breaks existing code: the feature assumes that all variables from before the change, are non-nullable. To cope with it, static analysis for null safety can be selectively enabled with a compiler switch at the project level.

Developers can opt-in for nullability checking when they are ready to deal with the resulting warnings. Still, this is in their own best interest, as the warnings might reveal potential bugs in their code.

The switch is implemented as a property in the project file. The feature can be enabled by adding the following line to the first `PropertyGroup` element of the project file:

```
<Nullable>enable</Nullable>
```

Additionally, the feature can be enabled selectively inside an individual file by using the `#nullable` directive:

```

#nullable enable
// feature enabled
#nullable disable
// feature disabled
#nullable restore
// feature restored to project-level setting

```

C# already has support for iterators (explained in Chapter 44 – “How to implement

a method returning an `IEnumerable`?) and asynchronous methods (introduced in Chapter 63 – “What is the recommended asynchronous pattern in .NET?”).

In C# 8.0, the two are combined into **asynchronous streams**. They are based on asynchronous versions of the `IEnumerable` and `IEnumerator` interfaces:

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken
cancellationTok = default);
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }

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

Additionally, an asynchronous version of the `IDisposable` interface is required for consuming the asynchronous iterators:

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

This allows the following code to be used for iterating over the items:

```
var asyncEnumerator = GetValuesAsync().GetAsyncEnumerator();
try
{
    while (await asyncEnumerator.MoveNextAsync())
    {
        var value = asyncEnumerator.Current;
        // process value
    }
}
```

```
finally
{
    await asyncEnumerator.DisposeAsync();
}
```

It's very similar to the code we're using for consuming regular synchronous iterators. However, it does not look familiar because we typically just use the **foreach** statement instead. An asynchronous version of the statement is available for asynchronous iterators:

```
await foreach (var value in GetValuesAsync())
{
    // process value
}
```

Just like with the **foreach** statement, the compiler generates the required code itself.

It's also possible to implement asynchronous iterators using the **yield** keyword, similar to how it can be done for synchronous iterators:

```
private async IEnumerable<int> GetValuesAsync()
{
    for (var i = 0; i < 10; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

Cancellation tokens are also supported with this syntax. The **EnumeratorCancellation** attribute can be used to annotate the parameter which will receive the **CancellationToken** passed to the **GetAsyncEnumerator** method:

```
private async IEnumerable<int>
GetValuesCancellableAsync([EnumeratorCancellation]
CancellationToken token = default)
{
    for (var i = 0; i < 10; i++)
    {
        await Task.Delay(1000, token);
    }
}
```

```

        yield return i;
    }
}

```

When using `await foreach` with such an asynchronous iterator, the `CancellationToken` can be passed to the `GetAsyncEnumerator` method by using the `WithCancellation` extension method:

```

await foreach (var value in GetValuesCancellableAsync().
WithCancellation(token))
{
    // process value
}

```

LINQ methods for the new `IAsyncEnumerable<T>` interface are available in the standalone `System.Interactive.Async` NuGet package which is a part of the `Reactive Extensions` project.

Before C# 8.0, interfaces were not allowed to contain method implementations. They were restricted to method declarations:

```

public interface ISample
{
    void M1(); // allowed in C# 7
    void M2() => Console.WriteLine("ISample.M2"); // not allowed in
C# 7
}

```

To achieve similar functionality, abstract classes could be used instead:

```

public abstract class SampleBase
{
    public abstract void M1();
    public virtual void M2() => Console.WriteLine("SampleBase.M2");
}

```

In spite of this, C# 8.0 added support for **default interface methods**, i.e. method implementations using the syntax in the first example above. This allows scenarios not supported by abstract classes.

A library author can now extend an existing interface with a default interface method implementation, instead of a method declaration.

This has the benefit of not breaking existing classes, which implement the old version of the interface. If they don't implement the new method, they can still use the default interface method implementation. When they want to change that behavior, they can override it, but no code change is required just because the interface has been extended.

Since multiple inheritance is not allowed, a class can only derive from a single base abstract class.

In contrast to that limitation, a class can implement multiple interfaces. If these interfaces implement default interface methods, this effectively allows classes to compose behavior from multiple different interfaces – this concept is known as [traits](#) and is already available in many programming languages.

Some pattern matching features have already been added to C# in version 7.0. The support has been further extended in C# 8.0.

Three new pattern types have been added:

- **Positional patterns** allow deconstruction of matched types in a single expression. They depend on the **Deconstruct** method implemented by a type (you can read more about the Deconstruct method in Chapter 79 – “How did tuple support change with C# 7?”):

```
if (sword is Sword(10, var durability)) {
    // code executes if Damage = 10
    // durability has value of sword.Durability
}
```

- **Property patterns** are similar positional patterns but don't require the Deconstruct method. As a result, the syntax to achieve equivalent functionality to the example above is a bit longer because it must explicitly specify the property names:

```
if (sword is Sword { Damage: 10, Durability: var durability })
{
    // code executes if Damage = 10
}
```

```
        // durability has value of sword.Durability
    }
```

- **Tuple patterns** allow matching of more than one value in a single pattern matching expression:

```
switch (state, transition)
{
    case (State.Running, Transition.Suspend):
        state = State.Suspended;
        break;
}
```

Additionally, **an expression version of the switch statement** allows terser syntax when the only result of pattern matching is assigning a value to a single variable:

```
state = (state, transition) switch
{
    (State.Running, Transition.Suspend) => State.Suspended,
    (State.Suspended, Transition.Resume) => State.Running,
    (State.Suspended, Transition.Terminate) => State.NotRunning,
    (State.NotRunning, Transition.Activate) => State.Running,
    _ => throw new InvalidOperationException()
};
```

The discard character (`_`) is used for the default case. If it's not specified in the expression and the value doesn't match any of the other cases, a **SwitchExpressionException** will be thrown.

88

Q88. How are .NET Core and .NET Standard evolving?

Ever since I started working on this book in early 2018, the development of .NET Core has continued with full speed. Fast forward to the present day, we find several important releases were made available which aren't covered in the other chapters.

This chapter was last updated for the release of .NET Core 3.0 in September 2019 which was a major milestone in the development of .NET Core.

Note: You can read about the difference between the .NET framework and .NET Core in Chapter 2 – "How is .NET Core different from the .NET framework?".

In May 2018, **.NET Core 2.1** was released. Its main focus was performance:

- Build times were drastically improved. Build time of a large project got 10 times faster than before.
- A new memory efficient type, `Span<T>`, was introduced which provides an isolated view of a part of a larger array without any memory allocations. It is very useful for efficient parsing and other processing of large inputs.

- `Span<T>` was used to implement a new `HttpClient` handler class for better networking performance.
- A lot of work was done on the JIT (just-in-time) compiler to improve the runtime performance of applications.

New features were also introduced. The most important ones were as follows:

- The Windows compatibility pack included 20 thousand APIs from the .NET framework which are not included in .NET Core itself (e.g. the `System.Drawing` namespace, support for Windows services, the registry APIs, the `EventLog` class etc.). This pack could primarily be used to make porting of existing .NET Framework applications to .NET Core easier. Despite its name, it's not just available only for Windows. About half of the APIs are also implemented on other platforms. The rest throw an exception when invoked on a non-Windows OS.
- .NET Core Tools represent a new (NuGet based) way for deploying command line (CLI) tools implemented as .NET Core console applications. They are modelled after the NPM (Node Package Manager) global tools. This infrastructure should make it easier to develop and distribute small executables to be run directly from the command line or from a script. A list of available tools is maintained on [GitHub](#). Most of them are designed to be used during development and building of applications (e.g. code generators, static code analyzers, build tools, etc.).

To install a simple HTTP server named `dotnet-serve`, the following command can be used:

```
dotnet tool install -g dotnet-serve
```

It will be added to the path (i.e. the `PATH` environment variable will be modified) so that it can later be invoked using only its name:

```
dotnet-serve
```

In December 2018, **.NET Core 2.2** was released. It didn't have as many improvements as .NET Core 2.1. Most changes were introduced to ASP.NET Core, e.g.:

- Better support for Open API (Swagger) descriptors for HTTP (REST) services. Customizable conventions were added as a simpler alternative to individual documentation attributes. A diagnostic analyzer is included for detecting

mismatches between the code and the documentation attributes (you can read more about diagnostic analyzers in Chapter 5 – "What is the Roslyn compiler and how can I use it?").

- A new router with improved performance and a better globally available link generator.

Entity Framework Core was also expanded with support for spatial data.

Although .NET Core 2.2 was released after .NET Core 2.1 and includes several new features, it's still recommended to use .NET Core 2.1 for most applications because it has a different support policy.

.NET Core 2.1 was released as an **LTS (Long Term Support) version** which means that it will be supported for 3 years after its initial release (August 2018 when .NET Core 2.1.3 was released which is the minor version that started the LTS lifecycle) or one year after the release of the next LTS version (whichever happens later).

In contrast, .NET Core 2.2 was released as a **Current version**. As such, it will only be supported for 3 more months after the next version of .NET Core is released (either Current or LTS). The 3-month maintenance period for .NET Core 2.2 started with the release of .NET Core 3.0 in September 2019. **This means that .NET Core 2.2 will only be supported until December 23, 2019.**

.NET Core 3.0 was also released as a Current version. The LTS version will be .NET Core 3.1 which is planned for release in November 2019.

.NET Core 3.0 is the first .NET runtime to **fully support C# 8.0**. You can read more about the new language features that it introduces in the previous Chapter 87 – "Which major new features does C# 8 bring?".

The so-called **.NET Core Global Tools** that were first introduced in .NET Core 2.1 can also be installed as **local tools** in .NET Core 3.0. When installed globally, they are always available from the command line of that computer. When installed locally as a part of a .NET Core project or solution, they are automatically available to all developers working on the same code repository.

Since further development of the .NET framework stopped with version 4.8, .NET Core 3.0 has a big focus on **Windows-specific features** which were previously not supported.

The goal is to make .NET Core a way forward for Windows developers who currently

use the .NET framework. Unlike almost all the other parts of .NET Core so far, these new features are not cross-platform and only work on Windows because they heavily rely on services provided by the operating system.

The most prominent new feature in .NET Core 3.0 is support for development of **Windows desktop applications** using Windows Forms and WPF (Windows Presentation Foundation). Both application models are fully compatible with their .NET framework counterparts. Therefore, any existing .NET framework applications using them can be ported to .NET Core unless they depend on some other feature that is not supported in .NET Core, such as .NET Remoting or advanced WCF features for example.

[Porting existing .NET framework applications to .NET Core](#) is not trivial and requires some changes to the source code and testing afterwards. Because of this, it only really makes sense for applications which are still being actively developed. Also, at the time of release, Visual Studio 2019 included only the WPF designer for .NET Core projects. The Windows Forms designer was still actively developed and only available in preview as an extension.

Although Windows Forms and WPF in .NET Core 3.0 are mostly just ports from .NET framework, there are two new features worth mentioning:

Porting existing .NET framework applications to .NET Core is not trivial and requires some changes to the source code and testing afterwards. Because of this, it only really makes sense for applications which are still being actively developed. Also, at the time of release, Visual Studio 2019 included only the WPF designer for .NET Core projects. The Windows Forms designer was still actively developed and only available in preview as an extension.

Although Windows Forms and WPF in .NET Core 3.0 are mostly just ports from .NET framework, there are two new features worth mentioning:

- Windows Forms has [enhanced support for high DPI screens](#). Since it is not fully compatible with high DPI behavior in the .NET framework, it requires additional testing and changes to the source code.
- Both Windows Forms and WPF support [XAML islands](#), i.e. the ability to host UWP (Universal Windows Platform) controls inside a Windows Forms or WPF application). This is particularly useful for (although not limited to) hosting first-party UWP controls such as WebView, Map Control, MediaPlayerElement and InkCanvas).

To simplify deployment of Windows desktop applications developed in .NET Core 3.0, the new application package format MSIX can be used. It's a successor to previous Windows deployment technologies, such as MSI, APPX and ClickOnce which can be used to publish the applications to Microsoft Store or to distribute them separately. A dedicated [Windows Application Packaging Project template in Visual Studio](#) is available to generate the distribution package for your application.

The significance of **COM (Component Object Model) components** has declined with increased adoption of the .NET framework. However, there are still applications which rely on it, e.g. automation in several Microsoft Office products is based on it.

.NET Core 3.0 applications on Windows can act [in both of the COM roles](#):

- As a COM client, activating existing COM components, for example to [automate a Microsoft Office application](#).
- As a COM server, [implementing a COM component](#) which can be used by other COM clients.

As part of .NET Core, a completely new version of **Entity Framework** was developed from scratch – **Entity Framework Core**. Although its API feels familiar to anyone who has used the original Entity Framework before, it is neither source code compatible nor feature equivalent to it. Any existing code using Entity Framework therefore needs to be rewritten in order to use Entity Framework Core instead.

To make porting of existing Windows applications using Entity Framework to .NET Core easier, .NET Core 3.0 includes a port of the original Entity Framework in addition to a new version of Entity Framework Core.

The **Entity Framework 6.3** version in .NET Core is primarily meant for porting existing applications. New applications should use Entity Framework Core instead which is still in active development.

At the time of release, the tooling for Entity Framework was only supported in Visual Studio 2019 on Windows. Designer support will be added in a later Visual Studio 2019 update. Until then, the models can only be edited from a .NET framework project referencing the same model.

However, applications developed with Entity Framework 6.3 for .NET Core can run on any platform which makes it suitable for porting not only Windows desktop applications but also web applications making them cross-platform in the process.

Unfortunately, [new providers are required for .NET Core](#). Initially only SQL Server provider is available. No support for SQL Server spatial types and SQL Server Compact is available or planned.

A new version of **Entity Framework Core** is also included with .NET Core 3.0.

[Among its new features](#), the most important are the improvements to LINQ which make it more robust and more efficient because larger parts of the queries are now executed on the database server instead of in the client application. As part of this change, [client-side evaluation of queries as fallback](#) when server-side query generation fails, was removed. Only the projection from the final **Select** part of the query might still be executed on the client.

This can break existing applications. Therefore, full testing of existing applications is required when upgrading to Entity Framework Core 3.0.

Additionally, Entity Framework Core 3.0 now includes support for [Cosmos DB](#) (implemented against its SQL API) and takes advantage of new language features in C# 8.0 (asynchronous streams). As a result, it can't be used with the .NET framework or older versions of .NET Core anymore.

ASP.NET Core is probably the most widely used application model in .NET Core as all types of web applications and services are based on it. Like Entity Framework Core, its latest version works only in .NET Core 3.0 and isn't supported in .NET framework.

To make applications smaller by default and reduce the number of dependencies, several libraries were removed from the basic SDK and must now be added manually when needed:

- Entity Framework Core must now be added to the project as a standalone NuGet package. A different data access library can be used instead.
- Roslyn was used for runtime compilation of views. This is now an optional feature which can be added to a project using [the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package](#).
- JSON.NET as the library for serializing and deserializing JSON has been replaced with [a built-in JSON library](#) with focus on high performance and minimal memory allocation. JSON.NET can still be installed into the project and used instead.

[Endpoint routing](#) which was first introduced in .NET Core 2.2 has been improved

further. Its main advantage is better interaction between routing and the middleware. Now, the effective route is determined before the middleware is run. This allows the middleware to inspect the route during its processing. This is especially useful for implementing authorization, [CORS \(Cross-Origin Resource Sharing\)](#) configuration and similar cross-cutting functionalities as middleware.

Probably the most heavily awaited new feature in ASP.NET Core 3.0 is **Blazor**. In .NET Core 3.0, only **server-side Blazor** is production ready (this feature was named Razor Components for a while in early previews of .NET Core 3.0).

Blazor makes client-side interactivity in a browser possible without any JavaScript code. If necessary, JavaScript can still be invoked (e.g. to use browser APIs, such as geolocation and notifications). All the other code is written in .NET instead.

In server-side Blazor, this code runs on the server and manipulates the markup in the browser using [SignalR](#). For this to work, constant connection between the browser and the server is required. Offline scenarios are not supported.

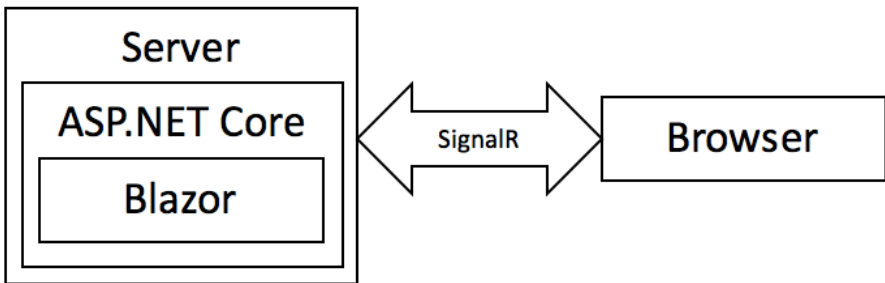


Figure 1: Blazor server-side execution model

Client-side Blazor was still in preview when .NET Core 3.0 was released and will ship later. As the name implies, all the code runs on the client in the browser like in JavaScript-based SPAs (single page applications). The .NET code is compiled to [web assembly](#) so that browsers can execute it.

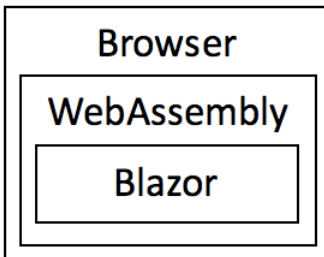


Figure 2: Blazor client-side execution model

Worker Service is a new project template for an ASP.NET Core application hosting long-running background processes instead of responding to client requests. There are support classes available to integrate such applications better with the hosting operating system:

- On Windows, the application can [act as a Windows Service](#).
- On Linux, it can [run as a systemd service](#).

[gRPC](#) is a modern high-performance contract-first RPC (remote procedure call) protocol developed by Google and supported in many languages and on many platforms. It uses [HTTP/2](#) for transfer and [Protocol Buffers](#) (also known as Protobuf) for strongly-typed binary serialization. This makes it a great alternative to WCF (Windows Communication Foundation) which has only limited support in .NET Core:

- [The client libraries](#) only support a subset of WCF bindings.
- There's only [an early open-source .NET Core port](#) of the server libraries available.

While Web API can be used to implement a REST service, gRPC is better suited to remote procedure call scenarios which were the most common use case for WCF services. Its performance benefits are most obvious when many RPC calls and large payloads are required.

Although a gRPC library for C# has been available for a while, .NET Core 3.0 and Visual Studio 2019 now include [first-class support](#) with project templates and tooling to make development easier. You can read more about [gRPC with ASP.NET Core 3.0](#).

.NET Core supports two deployment models:

- **Framework-dependent deployments** require that a compatible version of the .NET Core runtime is installed on the target computer. This allows the deployment package to be smaller because it only needs to contain compiled application code and third-party dependencies. These are all platform independent, therefore a single deployment package can be created for all platforms.
- **Self-contained deployments** additionally include the complete .NET Core runtime. This way the target computer doesn't need to have the .NET Core runtime preinstalled. As a result, the deployment package is much larger and specific to a platform.

Before .NET Core 3.0, only self-contained deployments included an executable. Framework-dependent deployments had to be run with the dotnet command line tool. In .NET Core 3.0, framework-dependent deployments can also include an executable for a specific target platform.

Additionally, in .NET Core 3.0, self-contained deployments support **assembly trimming**. This can be used to make the deployment package smaller by only including the assemblies from the .NET Core runtime which are used by the application.

However, dynamically accessed assemblies (through Reflection) can't be automatically detected which can cause the application to break because of a missing assembly. The project can be manually configured to include such assemblies, but this approach requires the deployment package to be thoroughly tested to make sure that no required assembly is missing.

Both deployment models now also support the creation of **single-file executables** which include all their dependencies (only third-party dependencies in framework-dependent deployments, also the .NET Core runtime in self-contained deployments). Of course, these executables are always platform specific.

In the field of performance, the following features focus on reducing the application startup-time:

- The **Two-tier JIT** (just-in-time) compiler has been available for a while, but with .NET Core 3.0, it is enabled by default, although it can still be disabled. To improve startup-time, two-tier JIT performs a faster lower quality compilation first and makes a slower second pass at full quality later when the application is already running.
- **Ready-to-run images** introduce AOT (ahead-of-time) compilation to .NET Core. Such deployment packages are larger because they include a native binary alongside the regular IL (intermediate language) assemblies which are still needed. Because there's no need for JIT compilation before startup, the application can start even faster. For now, there's no cross-targeting support for creating ready-to-run-images, therefore they must be built on their target platform.

A closer look at .NET Core 3.0 makes it clear that .NET Core is maturing.

Since there's no further development planned for .NET framework, **.NET Core is now**

the most advanced .NET implementation. It's cross-platform and more performant than .NET framework.

Most of the features from .NET framework are now also included. For exceptions which aren't planned to be supported in .NET Core, there are alternatives available:

- for Web Forms, there's Blazor
- for WCF and .NET Remoting, there's Web API and gRPC
- for Workflow Foundation, there's [Azure Logic Apps](#).

	Cross-platform	Windows only
.NET Core 2.0	console applications Entity Framework Core ASP.NET Core MVC ASP.NET Core Razor pages ASP.NET Core Web API	Windows Compatibility Pack
.NET Core 3.0	Entity Framework 6.3 Blazor Server-Side gRPC services worker services	Windows Forms WPF XAML islands COM support

Table 1: .NET Core application models and libraries

As a companion to .NET Core 3.0, **.NET Standard 2.1** was also released. In comparison to .NET Standard 2.0, it includes many new classes and methods which were added to .NET Core 3.0 making it possible for cross-platform class libraries to use them as well.

Despite that, it's still a good idea to write your libraries for .NET Standard 2.0 whenever possible because this will make them available to more different .NET runtimes, e.g. .NET framework (which isn't going to support .NET Standard 2.1) and previous versions of .NET Core. You can read more about .NET Standard in Chapter 3 – “What is .NET Standard and why does it matter to me?”

Thank you for reading this book and we hope you found it useful! Email us your feedback at

ebooks+csharp@a2zknowledgevisuals.com.

Join us at [Twitter](#) and [Facebook](#).