

Parallel Programming with **C#** — and — **.NET Core**

Developing Multithreaded Applications Using C# and
.NET Core 3.1 from Scratch



RISHABH VERMA
NEHA SHRIVASTAVA
RAVINDRA AKELLA

bpb

Parallel Programming with C# and .NET Core

*Developing Multithreaded Applications
Using C# and .NET Core 3.1 from Scratch*

Rishabh Verma
Neha Shrivastava
Ravindra Akella



FIRST EDITION 2020

Copyright © BPB Publications, India

ISBN: 978-93-89423-327

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990 / 23254991

DECCAN AGENCIES

4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967 / 24756400

MICRO MEDIA

Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296 / 22078297

BPB BOOK CENTRE

376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

Dedicated to

All the enthusiastic readers and to the wonderful .NET community!

*All the COVID-19 warriors in the world who are fighting the war
against this deadly virus tirelessly and risking their lives to save the
human race! More power to them!*

About the Authors



Rishabh Verma is a Microsoft certified professional and works at Microsoft as a senior development consultant, helping the customers to design, develop, and deploy enterprise-level applications. An electronic engineer by education, he has 12+ years of hardcore development experience on the .NET technology stack. He is passionate about creating tools, Visual Studio extensions, and utilities to increase developer productivity. His interests are .NET Compiler Platform (Roslyn), Visual Studio Extensibility, code generation, and .NET Core. He is a member of the .NET Foundation (<https://www.dotnetfoundation.org>). He occasionally blogs at <https://rishabhverma.net/>. His twitter id is @VermaRishabh, and his LinkedIn page is <https://www.linkedin.com/in/rishabhverma/>



Neha Shrivastava is a Microsoft certified professional and works as a software engineer for the Cloud & AI group at Microsoft India Development Center. She has about ten years' development experience and has expertise in the financial, healthcare, and e-commerce domains. Neha did a BE in electronics engineering. Her interests are the ASP.NET stack, Azure, and cross-platform development. She is passionate about learning new technologies and keeps herself up to date with the latest advancements. Her LinkedIn profile is <https://www.linkedin.com/in/neha-shrivastava-99a80135/>



Ravindra Akella works as a Senior Consultant at Microsoft with more than 13 years of software development experience. Specializing in .NET and web-related technologies, his current role involves end to end ownership of products right from architecture to delivery. He has lead software architecture, design, development, and delivery of large complex solutions with >80 software engineers using Azure Cloud and related technologies. He is a tech-savvy developer who is passionate about embracing new technologies. He has delivered talks and sessions on Azure and other technologies in international conferences. His LinkedIn profile is <https://www.linkedin.com/in/ravindra-akella/>

Acknowledgements

When a book gets published, only the names of authors and editors find a mention, but numerous unsung heroes play an equally important role, and without them, the project cannot be completed or successful. I have a long list of such heroes to be thanked.

My sincere gratitude to my architects (Ranjiv Sharma, Shrenik Jhaveri, Prasad Ganganagunta) as every discussion with them gave me a fresh perspective on a problem and something new to learn. My heartfelt thanks to my managers (Ashwani Sharma, Manish Sanga), who have always encouraged and supported me in writing this book apart from my professional work. A big shout out to all my colleagues, friends, and team members who made me feel good about book writing and supporting me.

Without reliable support from home, things appear rather challenging to accomplish, especially when they take away your time. I am grateful to my parents (Smt. Pratibha Verma & Shri R C Verma) and my brother (Rishi Verma) for their continued support and being a constant source of energy. I owe this book to my wife and co-author Neha, who sacrificed her numerous weekends and supported me in meeting the deadlines.

Lastly, but most importantly, I would like to thank my team (Neha, Ravindra) and the fantastic team of BPB Publications, for providing us with this opportunity to share our learning and contribute to the community.

—Rishabh Verma

I would like to acknowledge with my sincerest gratitude the support and love of my parents, Smt. Archana Shrivastava and Shri O.P. Shrivastava; my brother, Dr. Utkarsh Shrivastava, sister-in-law Dr. Samvartika Shrivastava and last but not the least, my husband, who is also my co-author in this book, Rishabh. They all kept me going with their constant support and encouragement.

—Neha Shrivastava

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. I am incredibly grateful to my parents for their love, prayers, caring and sacrifices, I am very much thankful to my wife Srividya and my son Vaarush for their love, understanding, prayers and continuing support to complete this book.

Finally, I would like to thank my co-author Rishabh and BPB publications for giving me this opportunity to write this book.

—Ravindra Akella

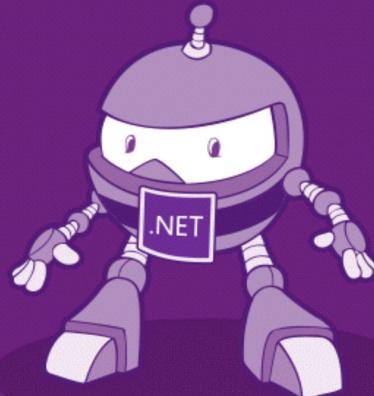


Independent. Innovative. And always open source.

The .NET Foundation is an independent organization created to foster innovation, which we believe starts with open development and collaboration. It's also a forum for community and commercial developers to broaden and strengthen the future of the .NET ecosystem.

The .NET Foundation supports .NET open source in a number of ways.

- Promote the broad spectrum of software available to .NET developers through NuGet.org, GitHub, and other venues.
- Advocate for the needs of .NET open source developers in the community.
- Evangelize the benefits of the .NET platform to a wider community of developers.
- Promote the benefits of the open source model to developers already using .NET.
- Offer administrative support for a number of .NET open source projects.



Services for .NET Foundation projects

The .NET Foundation provides several services to support the projects in our community. If you're currently running a project along these lines, or if you're interested in kicking off a new idea, please take a look at our New Project Checklist.

Management and Administration:

Project guidance and mentoring

IP and legal

Marketing and communications

Technical Services:

CLA Management

Software, Forums, AppVeyor

Secret Management

MyGet

SSL Certificates, Code Signing

Domain and DNS registration

See more details at <https://www.dotnetfoundation.org/about>

<https://dotnetfoundation.org/blog>

<https://twitter.com/dotnetfdn>

<https://www.meetup.com/pro/do.net/>

Preface

Application development has evolved over the last decade, and with the advent of the latest technologies like Angular, React on client-side, and ASP.NET Core, Spring on the server-side, the consumer expectations have risen like never before. The new mantra for software development these days is “Slow is the new downtime,” which means performance is one of the most crucial factors in application development and concurrency is one of the critical parameters that play a significant role in allowing applications to process requests simultaneously and hence improving perceived performance.

The primary objective of this book is to help readers understand the importance of asynchronous programming and various ways it can be achieved using .NET Core and C# 8 to build concurrent applications successfully. Along the way reader will learn the fundamentals of threading, asynchronous programming, various asynchronous patterns, synchronization constructs, unit testing parallel methods, debugging enterprise applications, and cool tips and tricks.

There are samples based on practical examples that will help the reader effectively use parallel programming. By the end of this book, you will be equipped with all the knowledge needed to understand, code, and debug multithreaded, concurrent and parallel programs with confidence.

Over the ten chapters in this book, you will learn the following:

Chapter 1: This chapter runs through the prerequisites to get started with the book. The chapter introduces several tools that help in working with parallel programming. We will also install Visual Studio 2019 and develop our very first sample C# 8 application on .NET Core 3.1 using Visual Studio 2019.

Chapter 2: This chapter introduces the readers with the new features and enhancements that are shipped in C# 8 with examples.

Chapter 3: .NET Core 3.1 is the latest and greatest major version of .NET Core. This chapter discusses the .NET Core 3.1 framework and describes what’s new in .NET Core 3.1.

Chapter 4: This chapter builds a solid foundation on parallel programming and demystifies the fundamental concepts and jargon that comes across while using

threads and tasks. The chapter also discusses the limitations of threads and tasks and when they should be avoided.

Chapter 5: This chapter introduces the concepts of data and task parallelism to the readers and discusses the new recommended async-await pattern in depth.

Chapter 6: In this chapter, we will take a deep dive on the patterns that are available using async-await and tasks which can be used in implementing enterprise application.

Chapter 7: In this chapter we will learn why synchronization is needed and various synchronization constructs and classes available in .NET Core 3.1

Chapter 8: Unit testing is one of the critical aspects of software development and even more so in multithreaded, concurrent, and parallel programming. In this chapter, we will see how to unit test asynchronous methods and various frameworks available to write useful unit tests.

Chapter 9: Debugging is an essential part of application development as well as bug fixing. This chapter discusses debugging multithreaded applications in detail and introduces various tools that can help you debug multithreaded applications in development as well as production environments.

Chapter 10: This chapter shares the tips, tricks, and best practices of multithreading and parallel programming with the readers.

Downloading the code bundle and coloured images:

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/362f5>

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

1. Getting Started	1
Structure	1
Objective.....	2
Download essential tools for Windows	2
Installing Visual Studio 2019 with .NET Core 3.1	4
Perfmon	4
Procmon.....	6
Process Explorer.....	9
PerfView	11
JustDecompile.....	13
DebugDiag	15
WinDbg.....	17
Creating a .NET Core 3.1 application using Visual Studio 2019	19
Summary	22
Exercise.....	22
2. What's New in C# 8?	23
Structure	23
Objective.....	24
C# 8 platform dependencies	24
New features and enhancements.....	25
<i>Nullable reference types/Non-nullable reference type</i>	25
<i>Asynchronous streams</i>	28
<i>Ranges and indices</i>	29
<i>System.Index</i>	30
<i>System.Range</i>	30
Default implementations of interface members	33
Readonly members on structs	35
Pattern matching enhancements.....	35
<i>Switch expressions</i>	35
<i>Recursive patterns</i>	38

<i>Positional pattern</i>	39
<i>Property pattern</i>	39
<i>Tuple patterns</i>	41
Using declarations	42
Static local functions	43
Disposable ref structs	46
Null-coalescing assignment.....	47
Interpolated verbatim strings enhancement.....	48
Summary	48
Exercise.....	49
3. .NET Core 3.1	51
Introduction	51
Structure	52
Objective.....	53
New features and enhancements.....	53
.NET Core version APIs	53
Windows Desktop application support.....	54
<i>Windows Desktop Deployment MSIX</i>	57
COM-callable components – Windows Desktop.....	57
WinForms high DPI	57
.NET Standard 2.1	58
C# 8 and its new features support.....	58
Compile and Deploy	58
<i>Default executable</i>	58
<i>Single executable file</i>	59
<i>Assembly linking</i>	60
<i>Tiered compilation</i>	60
<i>ReadyToRun images</i>	61
<i>Cross-platform/architecture restrictions</i>	61
Runtime/SDK.....	61
<i>Build copies dependencies</i>	62
<i>Local tools</i>	62
<i>Smaller Garbage Collection heap sizes</i>	63

<i>Garbage Collection Large Page supports</i>	63
<i>Opt-in feature</i>	63
IEEE Floating-point improvements.....	64
Built-in JSON support	64
<i>Json Reader</i>	65
<i>Json Writer</i>	68
<i>Json Serializer</i>	70
HTTP/2 support	71
Cryptographic Key Import and Export	71
Summary	72
Exercise.....	72
4. Demystifying Threading.....	73
Structure	73
Objectives	74
Why threading?.....	74
What is threading?	75
Thread.....	77
Exception handling.....	86
<i>Limitations</i>	87
ThreadPool.....	88
<i>Exceptions in ThreadPool</i>	94
<i>Limitations of Thread Pool</i>	94
ThreadPool in action.....	95
Task	95
TaskCreationOptions	98
Exception handling with Tasks	102
Cancellation	104
Continuations	108
WhenAll, WhenAny	110
Task Scheduler.....	111
Task Factory	114
Summary	114
Exercise.....	114

5. Parallel Programming	117
Structure	117
Objectives	118
Understanding the jargon.....	118
Parallel Extensions.....	119
Task Parallel Library (TPL).....	121
Data parallelism	123
Task parallelism.....	129
<i>PLINQ</i>	130
Data structures for parallelism.....	131
IEnumerator and yield return	141
async await.....	142
async await – Control flow	151
async await – Under the hood.....	153
<i>Language features</i>	153
Principles for using async await	157
Restrictions on async await	158
<i>CPU (compute) bound versus I/O bound work</i>	159
Deadlock.....	160
Asynchronous Streams.....	160
ValueTask	162
Summary	165
Exercise.....	166
6. The Threading Patterns	167
Introduction	167
Structure	167
Objectives	168
Task-based Asynchronous Pattern (TAP).....	168
Implementing pattern	168
CPU bound versus I/O bound	170
<i>Exception handling</i>	174
<i>Nested exception handling</i>	177
<i>Exception handling in Task.Wait()</i>	180
<i>Using the handle method</i>	182

<i>Avoid async void</i>	184
<i>Cancellation</i>	188
<i>Progress reporting</i>	195
<i>Other asynchronous patterns</i>	204
Asynchronous Programming Model (APM)	205
<i>APM to TAP wrapper</i>	208
<i>TAP to APM wrapper</i>	212
Event-based Asynchronous Pattern (EAP)	216
<i>EAP to TAP wrapper</i>	223
Summary	228
Exercise	229
7. Synchronization Constructs	231
Structure	232
Objectives	232
Overview	232
<i>Thread safety</i>	236
Locking constructs	236
<i>Lock or Monitor.Enter/Monitor.Exit (Exclusive)</i>	237
<i>Mutex (Exclusive)</i>	239
<i>SpinLock (Exclusive)</i>	243
<i>Semaphore (Non-Exclusive)</i>	245
<i>SemaphoreSlim (Non-exclusive)</i>	251
<i>Reader/Writer locks (Non-Exclusive)</i>	253
Signaling constructs	260
<i>AutoResetEvent</i>	261
<i>ManualResetEvent/ManualResetEventSlim</i>	266
<i>CountdownEvent</i>	271
<i>Barrier classes</i>	274
<i>Wait and Pulse</i>	279
<i>Interlocked class</i>	280
<i>Volatile class</i>	281
Summary	281
Exercise	281

8. Unit Testing Parallel and Asynchronous Programs	283
Structure	283
Objectives	283
Overview	284
Basics of unit testing with XUnit	285
<i>Executing unit tests</i>	286
<i>IntelliTest</i>	288
<i>Live Unit Testing</i>	290
<i>Unit test async methods</i>	290
<i>Unit test exceptions in async methods</i>	293
<i>Unit test async method using mock data</i>	294
Unit test for parallel methods	299
<i>Unit test async void methods</i>	310
Summary	310
Exercise	311
9. Debugging and Troubleshooting	313
Structure	314
Objectives	314
Debugging primer with Visual Studio 2019	314
Profiling	327
Memory Dumps	331
<i>Collecting memory dumps</i>	332
<i>Analyzing memory dumps</i>	342
Fixing	354
Performance analysis with PerfView	355
Summary	359
Exercise	359
10. Tips and Tricks	361
Structure	362
Objectives	362
Tips and tricks	362
<i>Threading and TPL</i>	363
<i>async await</i>	363

<i>ASP.NET Core</i>	365
<i>Threading Patterns</i>	366
<i>Synchronization</i>	366
<i>Testing</i>	367
<i>Debugging</i>	368
<i>Azure</i>	373
<i>Summary</i>	376

CHAPTER 1

Getting Started

"The secret to getting ahead is getting started!"

- *Anonymous*

As the name of chapter states, we will set up the required tools and get started with our journey of parallel programming with .NET Core 3.1 and C# 8. There are essential framework and tools to be downloaded and installed to start our learning and practical implementation on .NET Core 3.1 using windows operating system. We will begin the journey with the installation of Visual Studio 2019 and the latest version of .NET Core 3.1. We will create our first .NET Core 3.1 application. Though .NET Core 3.1 is cross-platform, we will focus the discussion on Windows as that's the most popular and widely used operating system platform on the planet. So, let's get started.

Structure

We will cover the following topics:

- Download essential tools for Windows
- Installing Visual Studio 2019 with .NET Core 3.1
- PerfMon
- ProcMon

- Proc.exe
- PerfView
- JustDecompile
- DebugDiag
- WinDbg
- Create your first .NET Core 3.1 application using Visual Studio 2019
- Summary
- Exercise

Objective

By the end of this chapter, the reader would:

- Learn to download and install all the required tools for .NET Core 3.1 and C# 8
- Create a “Hello World” application using .NET Core 3.1 template using Visual Studio 2019
- Learn to set up the development, debugging, troubleshooting and monitoring tools

Download essential tools for Windows

In this section, we will discuss the prerequisites. To have a seamless experience in learning .NET Core 3.1, we need to download and install a few developer tools. Microsoft recommends the Visual Studio Integrated Development Environment (IDE) to develop programs for Android, iOS, Windows applications, mobile applications, web applications, websites, web services, and cloud.

Navigate to the URL <https://visualstudio.microsoft.com/%20downloads/> in your preferred browser. Microsoft gives us options to select from 4 Visual Studio variations:

- **Community:** Powerful IDE, free for students, open-source contributors, and individuals, open-source contributors, and individuals
- **Professional:** Professional IDE best suited to small teams
- **Enterprise:** Scalable, end-to-end solution for teams of any size
- **Code:** The fast, free and open-source code editor that adapts to your needs

We can download one of these depending on our choice and description stated above. These descriptions are taken as-is from the download site. However, depending upon the selected option, you may or may not have features described in this book, like time travel debugging, and so on. For pure development purposes and following

the code snippets and samples of this book, Visual Studio 2019 Community version would suffice. The great thing is that this version is free. However, if it is possible for the reader, I would recommend Visual Studio 2019 Enterprise as it has a great set of tools and features for developing an enterprise-grade application. The authors of this book use Visual Studio 2019 Enterprise for code development and demonstration of tools.

We can go through Release Notes of each of the variants available to us to know more about them. Every variant and its small description are available on the above site:

The screenshot shows a web browser displaying the Microsoft Visual Studio download page at visualstudio.microsoft.com/%20downloads/. The page title is "Visual Studio 2019". Below the title, there are four download options:

- Visual Studio Enterprise 2019**: An integrated, end-to-end solution for developers looking for high productivity and seamless coordination across teams of any size. Please see the [Release notes](#) for more information. [Download ↓](#)
- Visual Studio Professional 2019**: Improve productivity with professional developer tools and services to build applications for any platform. Please see the [Release notes](#) for more information. [Download ↓](#)
- Visual Studio Team Explorer 2019**: A free solution for non-developers to interact with Azure DevOps Server and Azure DevOps. Please see the [Release Notes](#) for more information. [Download ↓](#)
- Visual Studio Community 2019**: A free, fully featured, and extensible solution for individual developers to create applications for Android, iOS, Windows, and the web. Please see the [Release notes](#) for more information. [Download ↓](#)

Figure 1.1: Download Visual Studio 2019

Visual Studio Code (VS Code): Apart from Community, Professional, and Enterprise variants of Visual Studio 2019, Microsoft provides a free open source code editor Visual Studio Code. It is a cross-platform code editor, and apart from Windows, VS code works with Linux and Mac OS. It's a cross-platform editor that can be extended with extensions available based on our requirements, for example, C# extension. It includes provision for embedded Git control, debugging, syntax highlighting, snippets, intelligent code completion, extensions support, and code refactoring.

Note: Visual Studio Code, like notepad, is an editor, and Visual Studio is an IDE. So Visual Studio Code is very lightweight, fast, with great support for debugging and has embedded Git control. It is a file and folders-based editor and doesn't need to know the project context, unlike an IDE. There is no **File | New Project** support in Visual Studio Code as we have in Visual Studio IDE. Instead, Visual Studio Code has a terminal, through which we can run .NET commands.

Apart from Visual Studio 2019, we are going to use few more tools in upcoming chapters like PerfMon for performance monitoring and troubleshooting, ProcMon tool for process monitoring, PerfView for performance analysis, and many more. We will see them in action in *Chapter 9, Debugging and Troubleshooting*.

Installing Visual Studio 2019 with .NET Core 3.1

For coding in Windows, as we discussed above, we can use:

- Visual Studio 2019 IDE
- Visual Studio Code editor

If we choose Visual Studio 2019, we just need to download Visual Studio 2019 version 16.4 or higher from <https://visualstudio.microsoft.com/vs/>. Visual Studio 2019 version 16.4 is the latest at the time of writing this chapter. It may change by the time; this book gets published. It comes with .NET Core 3.1 SDK and its project templates. So, we will be ready for development immediately after installing it.

Here I am installing Visual Studio Enterprise 2019. In **workloads** section, select **.NET Core cross-platform development**:

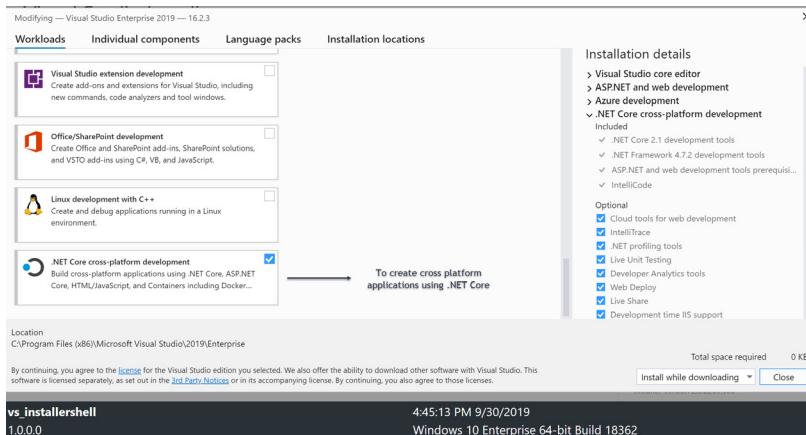


Figure 1.2: Select .NET Core cross-platform development workload

You can select additional workloads based on your needs. For this book, we need **.NET Core cross-platform development**, so we selected it and performed the installation. With this workload selected, the rest of the steps are straight forward, and the installation can be done without any issues.

Next, we will investigate other tools that we shall be using during this book. Let's start with the tool that comes installed by default in Windows Enterprise or Pro version of the operating system.

Perfmon

1. Perfmon, which is an acronym for performance monitoring, is the Windows reliability and performance monitoring tool. It helps us to troubleshoot the

issues, including application level to the hardware level. To open perfmon, go to **Run** (press *Windows key + R*), type *perfmon*, and then click the **OK** button:

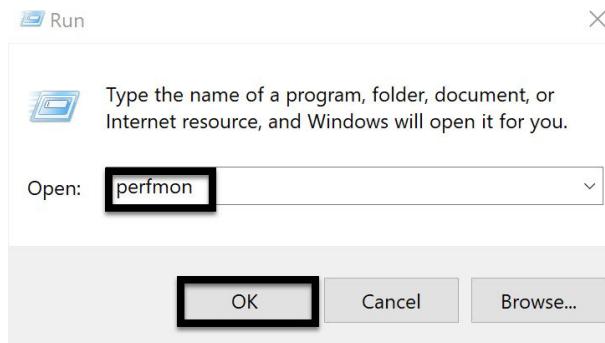


Figure 1.3: Open perfmon through Run or by clicking keys (*Windows + R*)

On the right pane of the performance monitoring tool, you can select graph type as line, histogram bar, or report. The detailed user guide can be seen in the tool in the **Help** menu item:

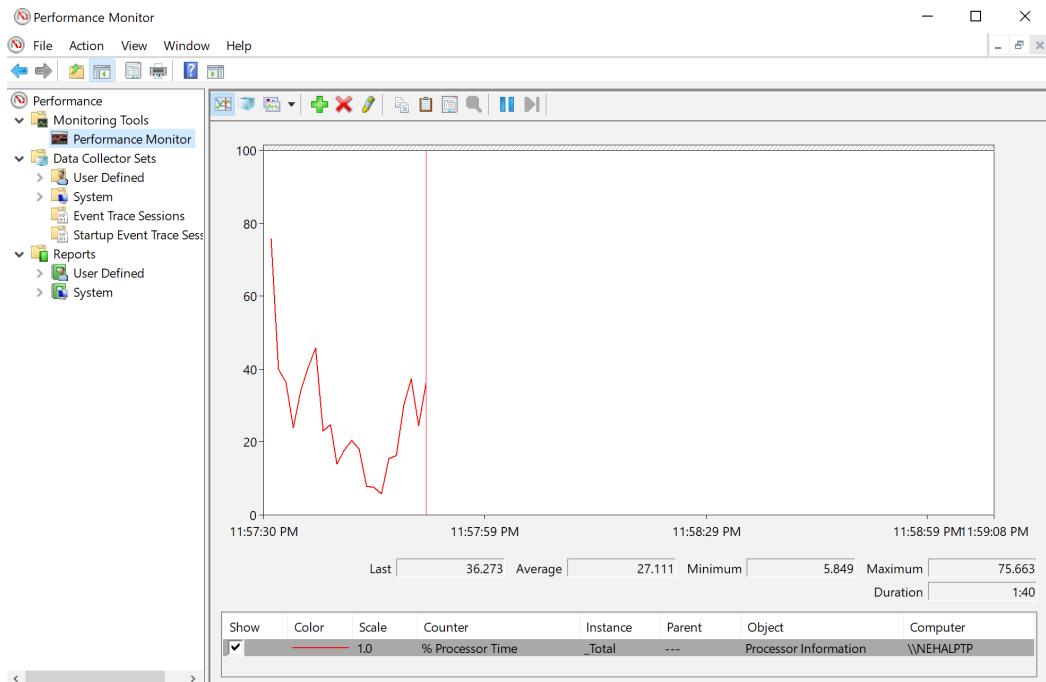


Figure 1.4: Performance monitoring graph view

On clicking **Performance** in the left pane, you can see an overview and summary which contains information about hardware, network, memory, and many more. Perfmon does an excellent job of collecting and displaying the Windows performance

counter data. The following is the screenshot from my laptop. We will discuss the use case of how to leverage Perfmon to monitor the performance counters of a .NET Core 3.1 application deployed on Windows in *Chapter 9*:

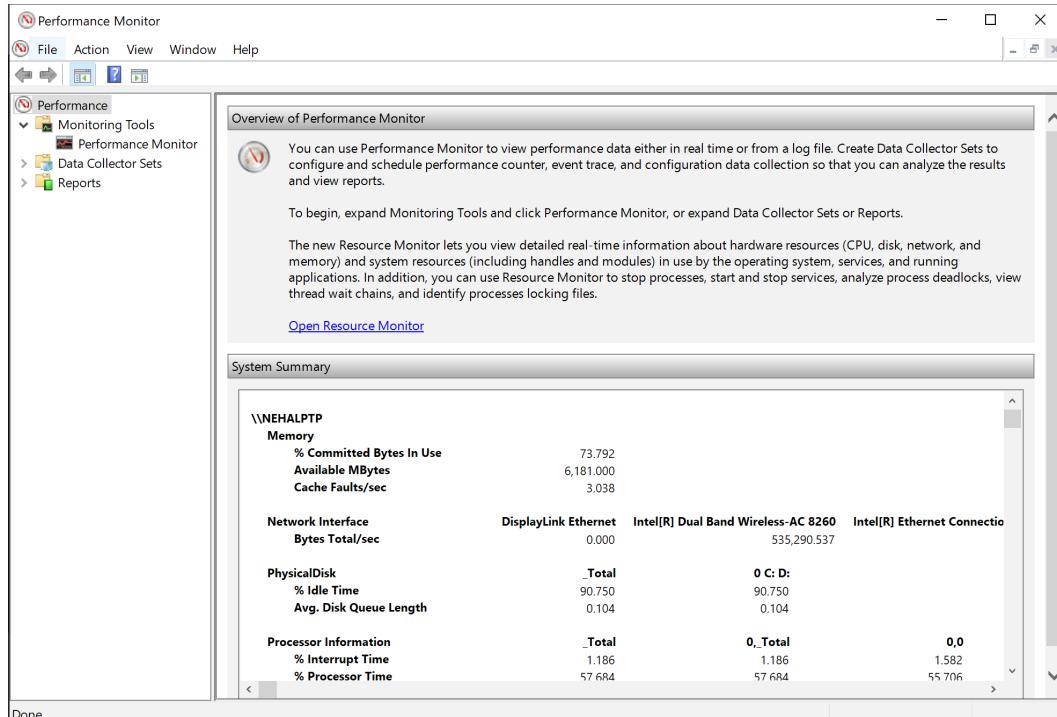


Figure 1.5: Overview of Perfmon and System Summary

Next, let us see tools, that we need to download, but do not require installation. These tools can be used by directly downloading them in your machine. Since these tools do not require any installation and work by simply copying/downloading them in machines, they can be used (if allowed) in the production servers and in impacted client machines for debugging or monitoring as appropriate.

Procmon

Process + monitor: Procmon is a process monitoring tool. Using this tool, we can find out what activities a process is doing on the registry, file system, network, threads, and so on. We can also see the loaded assembly modules and stack trace of the process, which helps to visualize what the process is doing. We will use Procmon in *Chapter 9*.

To download and run Procmon, please follow below steps:

1. Navigate to site <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon> and click on hyperlink **Download Process Monitor**:

The screenshot shows a web browser displaying the Microsoft Sysinternals website at <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon>. The page title is "Process Monitor v3.52". On the left, there's a sidebar with a "Filter by title" dropdown containing items like "Process Monitor", "PsExec", "PsGetSid", etc. The main content area has a "By Mark Russinovich" section, a "Published: March 24, 2019" date, and a "Download Process Monitor (1029 KB)" button. To the right, there's a "Is this page helpful?" poll with "Yes" and "No" options, and a "In this article" sidebar with links to "Introduction", "Overview of Process Monitor Capabilities", "Screenshots", "Related Links", and "Download".

Figure 1.6: Download procmon.exe

2. A file named **ProcessMonitor.zip** will be downloaded.
3. Unzip the file by right-clicking on the zip and then click on **Extract All**.
4. Open the extracted folder and optionally:
 - a. Copy the extracted **procmon.exe** to the path **%WINDIR%\System32** folder
 - b. Add the extracted folder path to the list of a **PATH** environment variable

Any of the preceding steps (a) or (b) would ensure that **procmon.exe** can be invoked from anywhere in the command prompt. In *Step a*, the **%WINDIR%** environment variable may resolve to **C:\Windows** path in most Windows machines. However, there may be cases in which the operating system is installed in other drive names, hence typing **%WINDIR%** would take you to the appropriate directory. Since the path **%WINDIR%\System32** is added to a Path environment variable, so Windows can locate **procmon.exe** in that path and run it whenever the command **procmon.exe** is executed on command prompt. In *Step b*, we just add the path of the extracted folder in the Path environment variable, and it has the same effect. Please note that both above steps are optional and if you just want to use it once, then you can skip them and just double click on **procmon.exe** to start using it, as described in next steps:
5. Double click on **procmon.exe**.
6. Click on the **Agree** button for agreement to the license terms. It appears only for the very first time and not always.

7. The process monitor would launch and display. We can now monitor the process that we want to.

As we discussed above, we can monitor multiple things. In the following screenshot, we can see **Tools** tab, where we can generate **Registry Summary**, **File Summary**, **Stack Summary**, **Network Summary**, and many more:

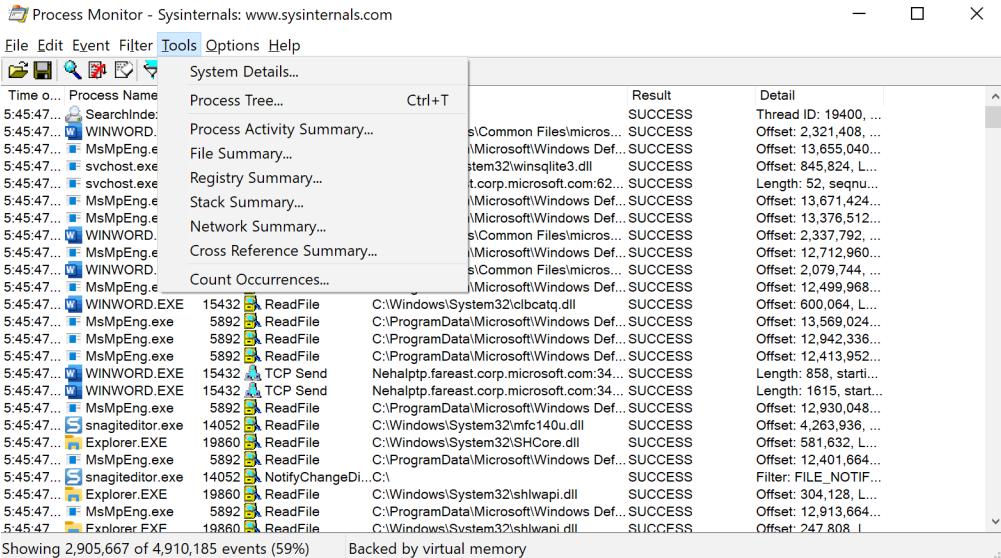


Figure 1.7: Process monitor and Tools tab

8. We can save these activities in a log file, by navigating to **File | Save**:

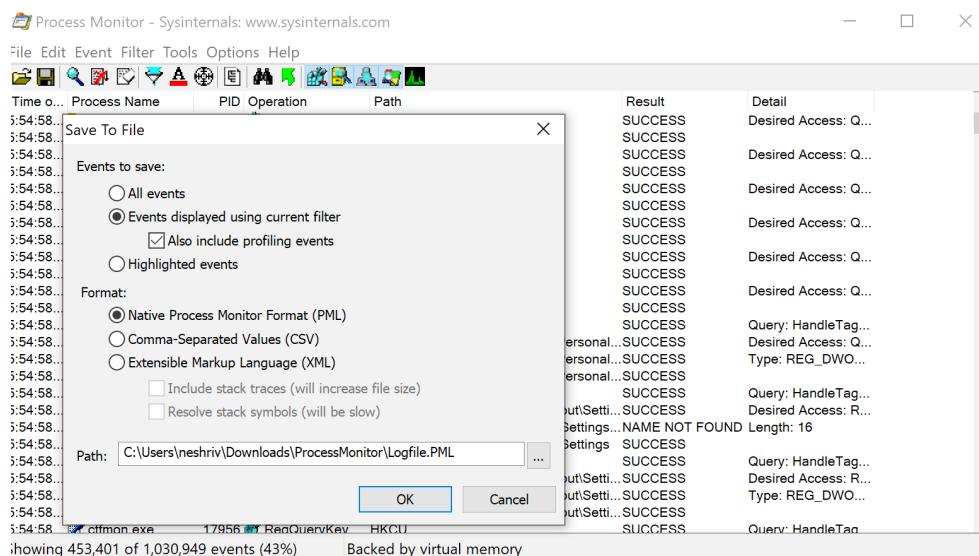


Figure 1.8: Save process activities in a log file

The **Help** menu provides you useful content to get started and know more about this tool.

Process Explorer

Process Explorer (`procexp`) is part of the Sysinternals toolkit. This tool displays all the processes and act as an advanced task manager and makes troubleshooting easy. It comes with an excellent search capability, and we can quickly get which process is loading what all DLLs and which process is locking a file or folder. We will see this tool in action in *Chapter 9*.

To download and run Process Explorer, follow the steps:

1. Navigate to site <https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer> and click on **Download Process Explorer**:

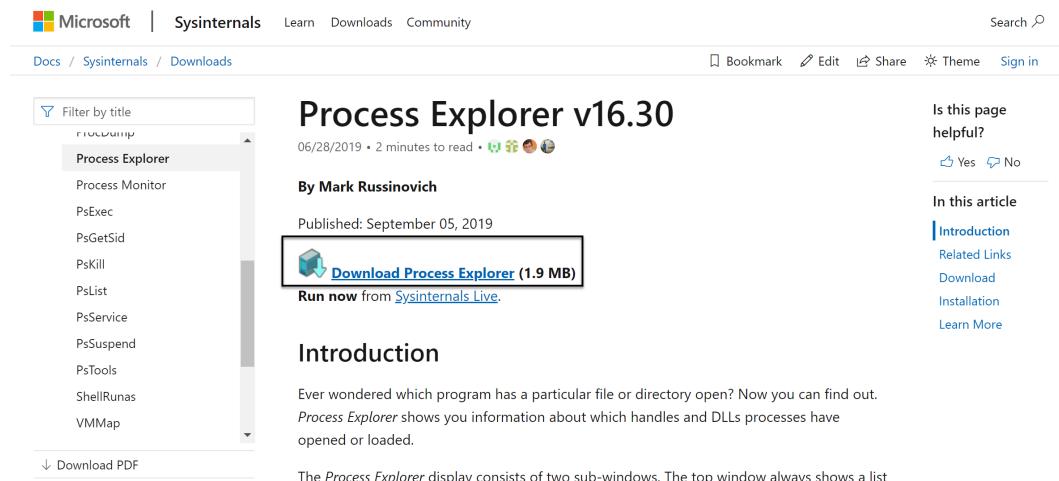


Figure 1.9: Download process explorer.exe

2. On clicking the above link, a file named `ProcessExplorer.zip` will be downloaded.
3. Unzip the folder by right-clicking on the zip and then clicking **Extract All** in the context menu.
4. Once it is extracted, open folder and double click on `procexp.exe`. You can also follow the optional steps mentioned for `procmon` in the preceding section if you intend to use the tool multiple times.
5. Click on the **Agree** button for agreement to the license terms. Again, this is just a one-time activity.
6. It will open the process explorer.

In the following screenshot from my machine, we can see all the processes and sub-processes in detail. If you click on a small graph image in the top, which is selected in a rectangle in the following screenshot, you can see the graph and details of CPU, Memory, I/O, GPU. It also displays the parent and child processes:

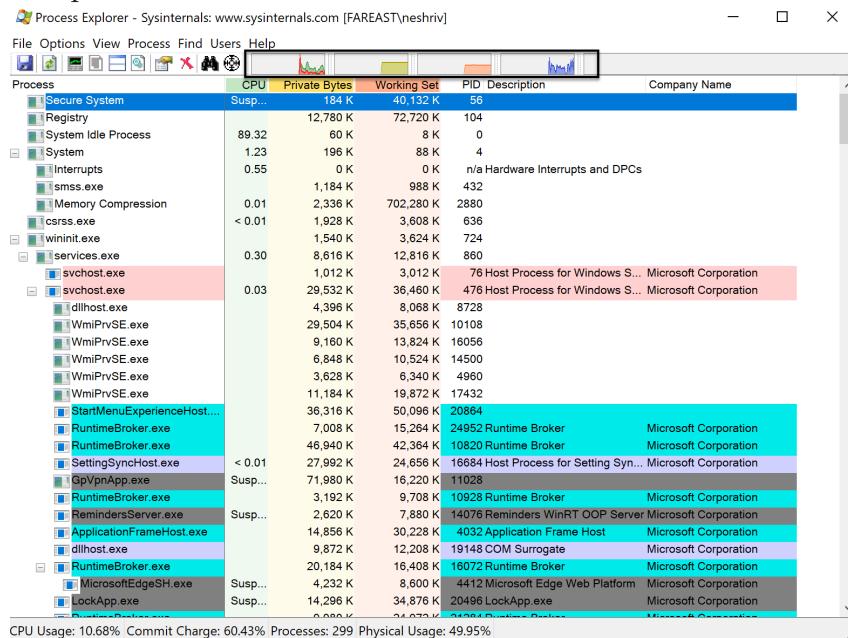


Figure 1.10: Process Explorer

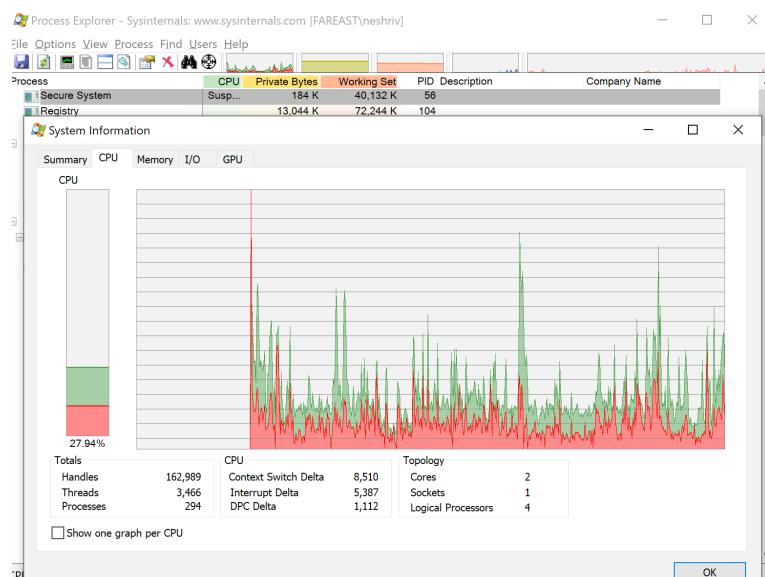


Figure 1.11: CPU graph

7. Click on **Find** menu item and then click on **Find Handler or DLL**. It will open a new window, enter the name which you want to search for. Process Explorer search will return all handlers and DLLs, which contains that name. A cool feature to find out what process is locking a file or DLL:

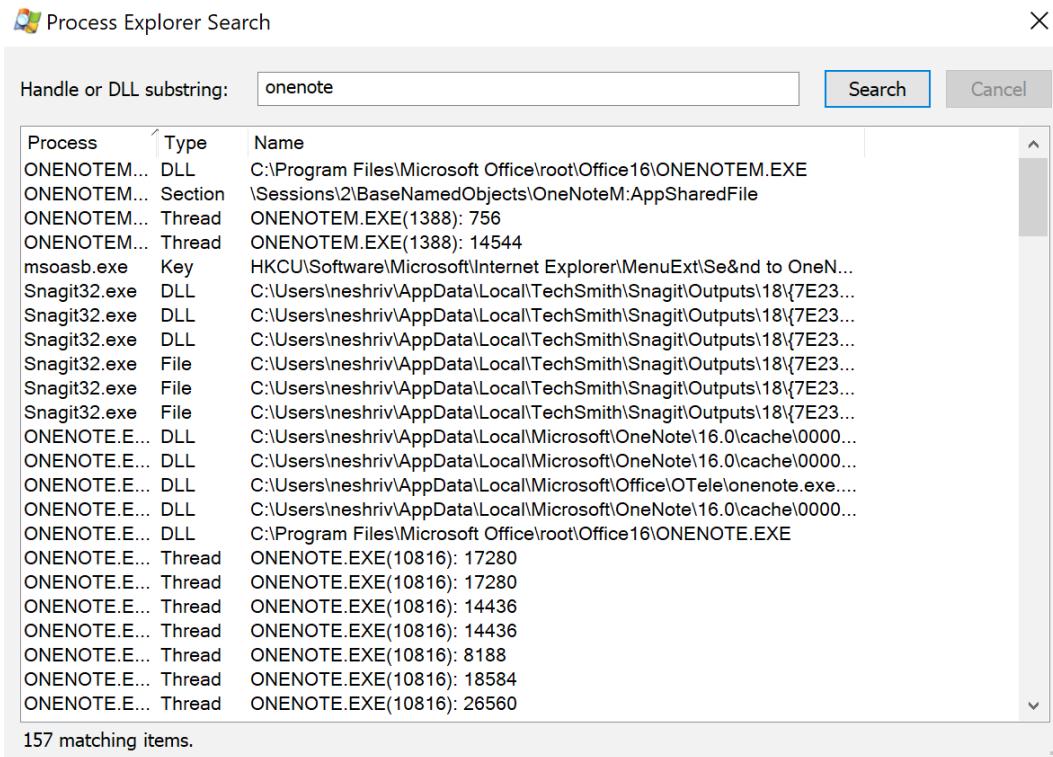


Figure 1.12: Process Explorer search

PerfView

PerfView, as the name suggests, is a tool for viewing and analyzing the performance of the process. It's a free tool for performance analysis from Microsoft and was developed by one of the architects in Microsoft to investigate performance issues. We will use it to perform a performance analysis of a .NET Core 3.1 application in Chapter 9. To understand its features in detail, go to <https://github.com/microsoft/perfview>.

To download PerfView, navigate to <https://github.com/microsoft/perfview/blob/master/documentation/Downloading.md>:

1. Click on **Download Version 2.0.43 of PerfView.exe** (<https://github.com/Microsoft/perfview/releases/download/P2.0.43/PerfView.exe>) to download an executable file. Please note that this version is the latest at the

time of writing this chapter. The version is subject to change, and website look and feel may also get updated in future, so the intention here is just to reach the documentation of PerfView and download the latest and greatest available version:

The screenshot shows a web browser window with the URL <https://docs.microsoft.com/microsoft/perfview/blob/master/documentation/Downloading.md>. The page title is "Downloading PerfView". Below the title, it says "PerfView is a free profiling tool from Microsoft. This page tells you how to get a copy of it for yourself. See the [PerfView Overview](#) for general information about PerfView." A section titled "PerfView Releases" follows, with a note that the GitHub Releases page is the official way to download versions of PerfView. A "Shortcut to Download the Latest PerfView.exe" section contains a link to "Download Version 2.0.43 of PerfView.exe". Below this, a note states that once clicked, the link will start downloading the executable file.

Figure 1.13: Download PerfView

2. Double click on **PerfView.exe** and click on **Run**.
3. Accept the license conditions (again, a one-time activity), and it will open PerfView for performance analysis:

The screenshot shows the PerfView application window. The menu bar includes File, Collect, Memory, Size, Help, Main View Help (F1), Tutorial, Troubleshooting, FAQ, Tips, Videos, and Feedback. The left pane shows a tree view of files under "C:\Users\neshriv\Downloads", with "ProcessExplorer" and "ProcessMonitor" expanded. The right pane displays the "Welcome to PerfView!" documentation. It explains that PerfView simplifies performance data collection and analysis. It covers collecting ETW Profile Data (CPU, Server Response Time, Wall Clock / Blocked Time, Unmanaged memory, Managed memory) and .NET Runtime Heap Data. It also describes viewing existing profile data and provides tips for using the tool. At the bottom, there are buttons for "Completed: View (Elapsed Time: 0.047 sec)", "Ready", "Log", and "Cancel".

Figure 1.14: PerfView tool

The usage of the PerfView tool would be discussed in *Chapter 9*.

JustDecompile

JustDecompile is a Telerik product, and it's a free tool, which efficiently decompiles any .NET/.NET Core assemblies like .dll, .exe, and many more and returns corresponding IL or C# code. To know more about JustDecompile features and function, go to the Telerik site: <https://www.telerik.com/products/decompiler.aspx>.

To install JustDecompile, follow below steps:

1. Navigate to <https://www.telerik.com/products/decompiler.aspx>.
2. Click **DOWNLOAD NOW** on top right corner:

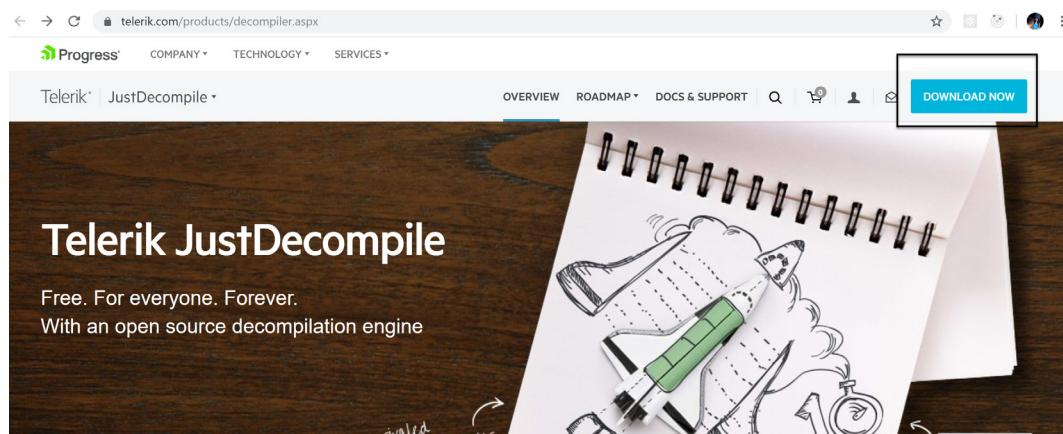


Figure 1.15: Download JustDecompile

The above Web UI is from the present-day site and is subject to change and you may or may not see the same UI, while you follow these steps:

1. **JustDecompileSetup.exe** will be downloaded.
2. Double click on **JustDecompileSetup.exe**, it will open installation wizard, check the checkbox for **JustDecompile** and click **Next**:

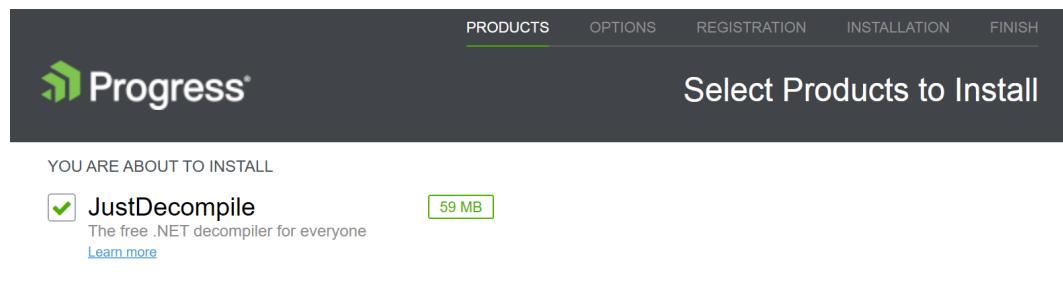


Figure 1.16: Select product to install

3. Select the installation folder location or leave it with a default value.
4. Check the checkbox for **Visual Studio Integration** if you want to integrate this tool with your Visual Studio.
5. Check the checkbox for **License agreement** and click **Next**:

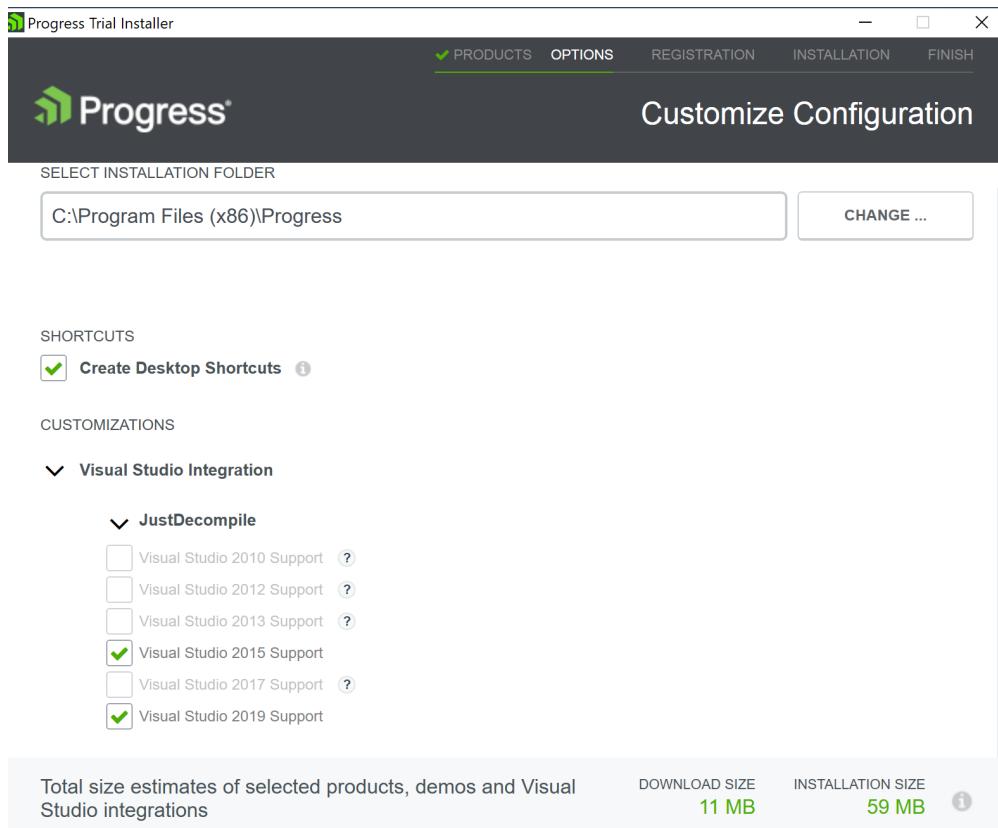


Figure 1.17: Select options for integration and location for installation

6. Here you will need to register with Telerik by providing the user details, or you can log in directly if you are already registered with Telerik.
7. Click **Next** and complete the installation.
8. JustDecompile is now ready to use.

We can use JustDecompile to see the IL and / or C# code of the managed .NET /.NET Core assemblies. Where we talk of decompiling or seeing the IL in the book, we will make use of the JustDecompile tool. The usage of the tool is discussed in *Chapter 5* and other chapters as needed.

Next, we will discuss the tools that require installation in the machine. These tools, therefore, may not be allowed to be used in production servers.

DebugDiag

DebugDiag is a debug diagnostic tool. This tool is useful to troubleshoot performance issues, memory leak related issues, and investigate application crashes and hangs. It is provided free of cost by Microsoft. We can analyze the memory dump file using this tool to do post-mortem debugging. Generally, memory dumps of the process are collected at the time abnormality or issue is discovered in the process. DebugDiag can be used to collect as well as analyze the collected memory dumps and find out the memory, threads, and CPU details, which can help debug the performance and memory leak related issues. It provides an excellent HTML report as output, which lists the analysis findings categorized as Information, Warning, and Errors. The great thing about this tool is that it is extensible, and we, as developers, can extend the rules or add new rules in DebugDiag to do repeated analysis for scenarios that are customized for our application.

To know more about this tool, visit the site <https://www.microsoft.com/en-us/download/details.aspx?id=58210>.

As of writing this chapter, the latest version of the DebugDiag tool is 2.3.0.37. It was released in April 2019.

Follow below steps to download and install **Debug Diagnostic Tool v2 Update 3**:

1. Open the **browser** and navigate to the site <https://www.microsoft.com/en-us/download/details.aspx?id=58210>.
2. Click on **Download** button
3. It will download **DebugDiagx64.msi**.
4. Double click on **DebugDiagx64.msi**, and it will open a setup wizard. Keep clicking **Next** and then click on **Install**. Please note that the book assumes that you are using Windows 10 Operating system, which is a 64-bit OS and hence, we are using the 64-bit version of the tool:

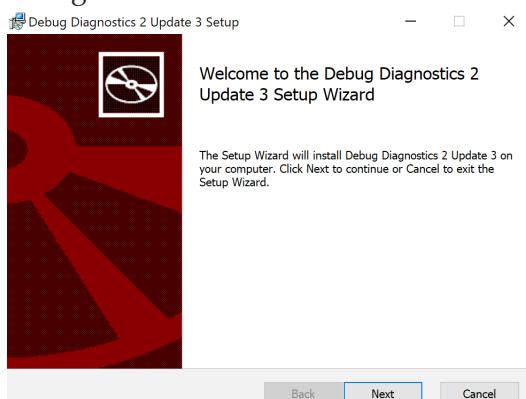


Figure 1.18: DebugDiag installation setup

5. After installation is done, click on **Finish**:

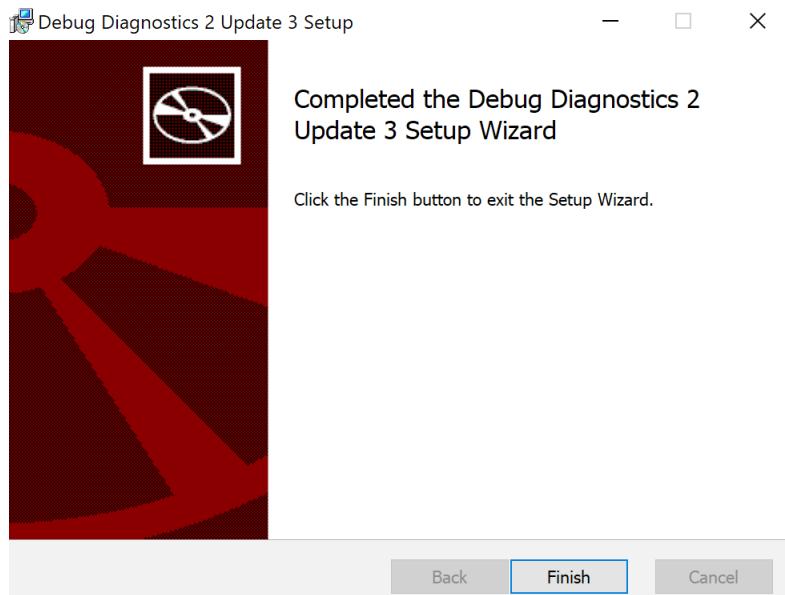


Figure 1.19: DebugDiag Installation

Once the installation is done, we can open this tool and can start analyzing **PerfAnalysis**, **CrashHangAnalysis**, **MemoryAnalysis**, **KernelCrashHangAnalysis**:

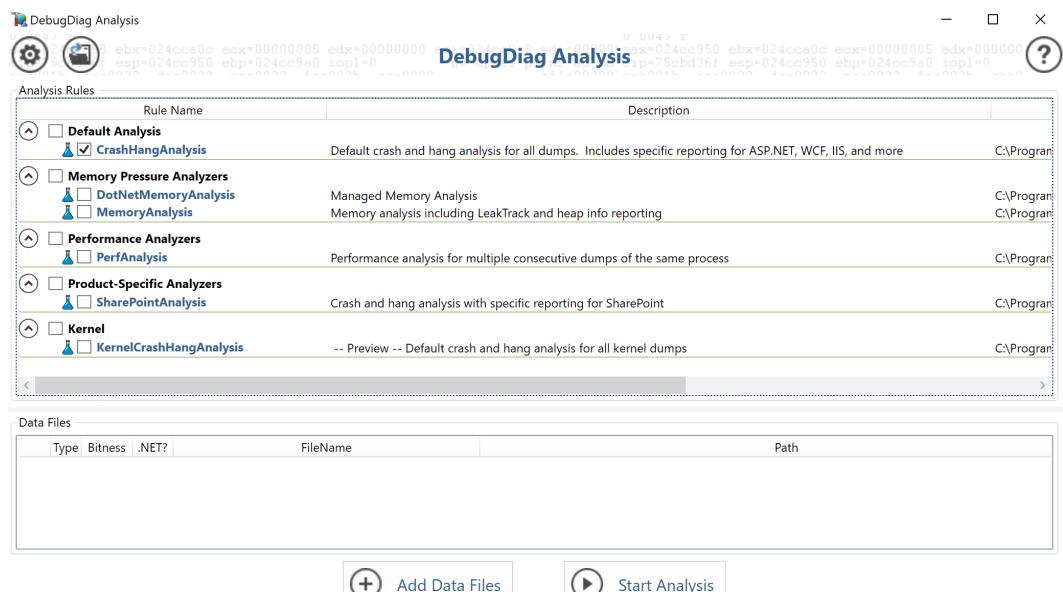


Figure 1.20: DebugDiag Analysis window

In the preceding screenshot, we see that we can select rules for which type of analysis we want to do, its brief description is given, and the location of the corresponding rule dll location is also displayed. Click on **Add Data Files** to load a memory dump file and then click on **Start Analysis** to start the analysis of the memory dump.

You can keep yourself updated with the latest and greatest version of the tool by enabling the following setting. Go to **Auto Update** tab and tick **Check for Updates on Startup** or **Automatically Install Updates on Startup without Confirming** checkboxes as shown in the following screenshot:

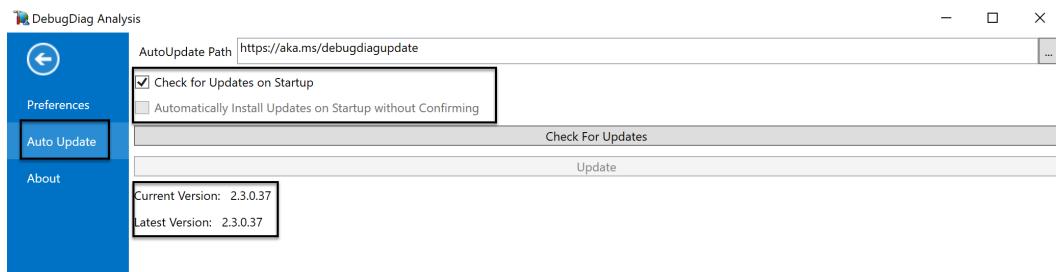


Figure 1.21: Update of DebugDiag

There is a DebugDiag collector as well as part of this download, which can be used to collect the memory dumps. We will see both the collection of memory dumps and analysis of memory dumps using DebugDiag in *Chapter 9*.

WinDbg

WinDbg is yet another great and powerful Microsoft product. It is a debugger useful to debug user or kernel mode, analyze crash dumps, and so on. It is a very advanced and powerful debugger and can be used for debugging memory dumps for both managed as well as unmanaged memory dumps. Microsoft support and product teams make extensive use of WindDbg for debugging and dump analysis. It is a command-based application and is slightly harder to use as compared to DebugDiag and needs some knowledge of commands to do the leak and performance analysis.

To learn more about WinDbg, go to Microsoft site <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools>. In this path you will find all feature details:

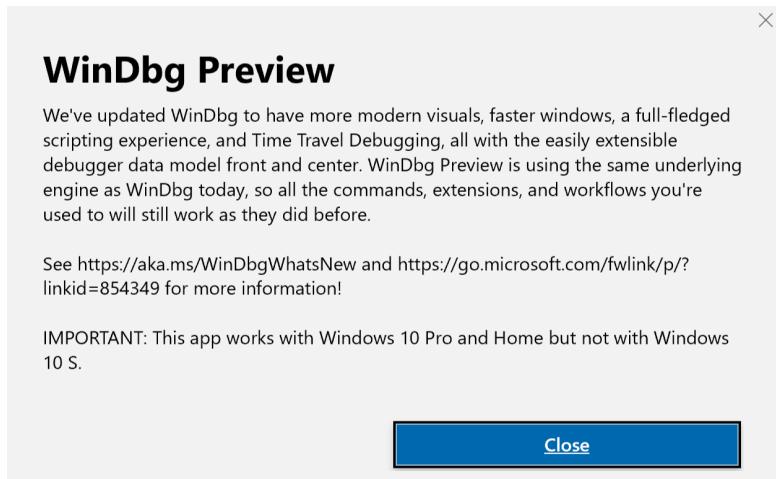


Figure 1.22: Information about WinDbg and its dependencies

1. Open Microsoft Store from your system and search WinDbg (search Microsoft Store in your Windows search).
2. Click on the button **Get**:

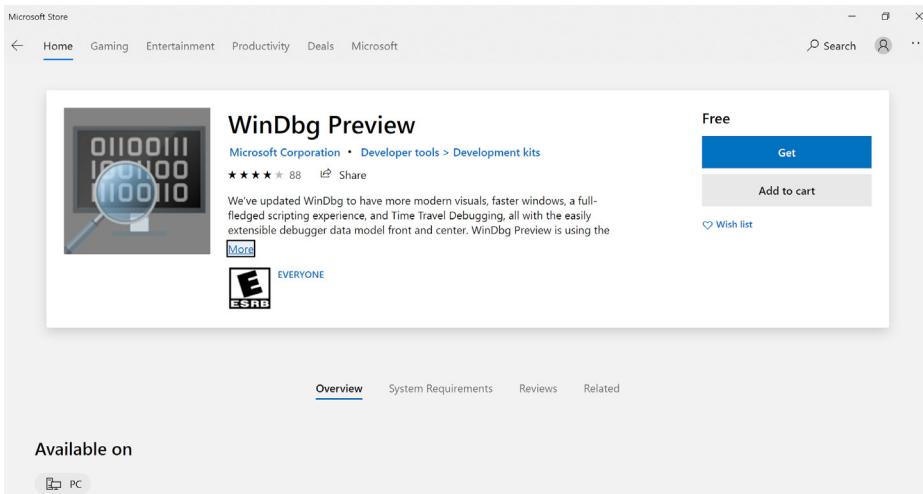


Figure 1.23: WinDbg Preview in Microsoft store

3. It will directly install WinDbg in your machine.
4. Once the installation is done, you can launch it.

It comes with many features and debugging options. We can debug executable by passing arguments; we can do time travel debugging or post-mortem debugging using this tool.

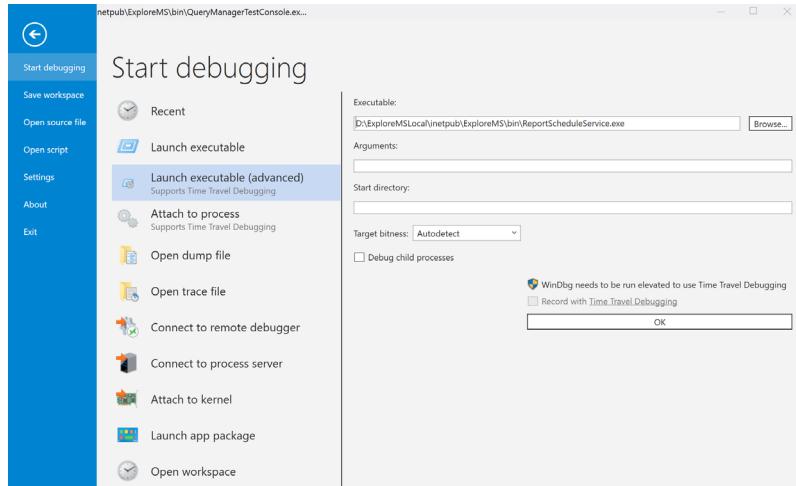


Figure 1.24: Start debugging using WinDbg

Creating a .NET Core 3.1 application using Visual Studio 2019

Let's start with the following steps:

1. Open Visual Studio 2019.
2. Go to **File | New | Project**. In the **New Project** dialog, you can see the .NET Core templates inside Visual C#:

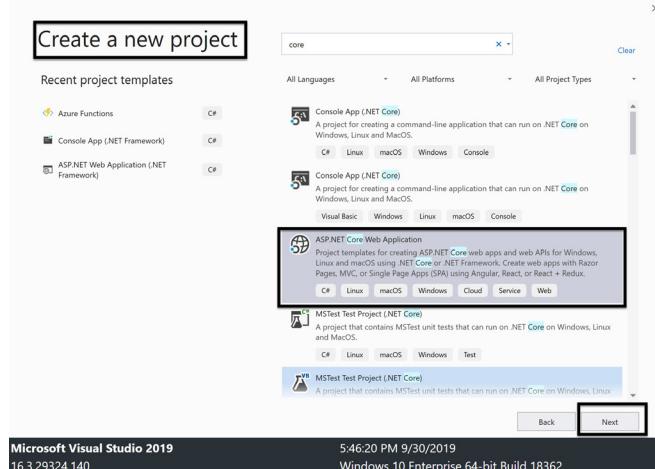


Figure 1.25: Create a new project and select ASP.NET Core Web App template

3. Click on **.NET Core** and select **ASP.NET Core Web Application**.
4. Name the project as DotNetCore31SampleApp or any other name of your choice and click **OK**.
5. It will show a new ASP.NET Core Web Application dialog. Ensure .NET Core and ASP.NET Core 3.1 is selected in the two dropdowns displayed in the following screenshot, as we are talking about .NET Core 3.1 here. The first dropdown implies the target framework of the application. We have the option to select the .NET Framework as the target framework or .NET Core in the first dropdown. If we select the .NET Framework, the application which we are going to create would not be cross-platform. If the application must run cross-platform, it should target .NET Core. The second dropdown is the version of ASP.NET Core that we are going to use.
6. The second dropdown has different versions of .NET Core, like 2.0, 3.0. We will keep it as ASP.NET Core 3.1. We can see multiple templates below and select one of them based on the requirement. We selected here **Web Application (Model-View-Controller)**. It also has advance features as **Configure for HTTPS** and **Enable Docker Support**. We checked **Configure for HTTPS** and kept **Enable Docker Support** unchecked as we are not going to use Docker. Click on **Create** for creating a new app:

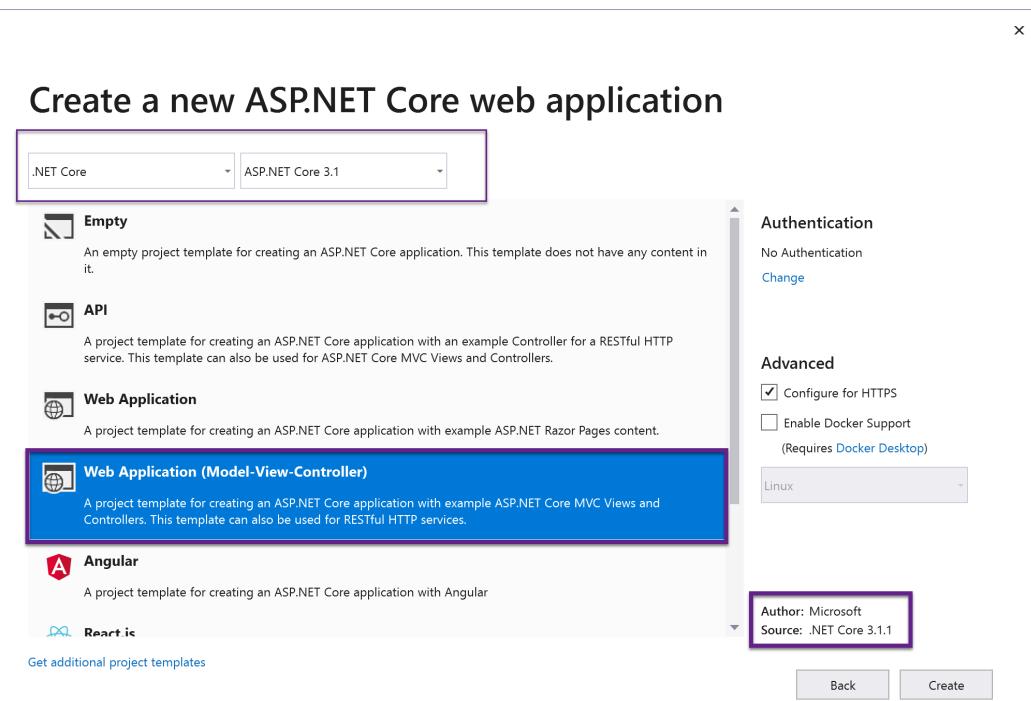


Figure 1.26: Select .NET Core version and project template as MVC

7. Yay! Visual Studio will create the `DotNetCore31SampleApp` project for us, and it will restore the essential packages to build in the background. We can see this by checking the Package Manager Console output:

Your very first ASP.NET Core 3.1 is ready to be run!

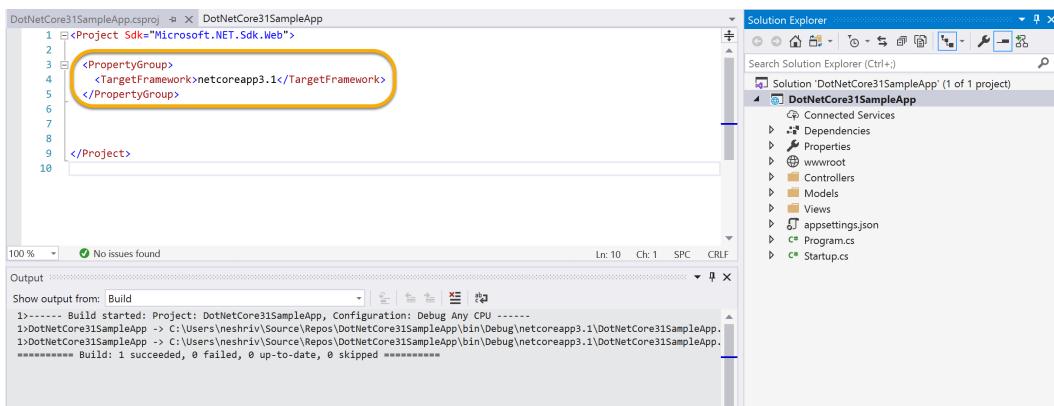


Figure 1.27: .NET Core 3.1 sample app solution window

8. Run this application, and it will open your default browser with default home page as shown in the following screenshot:

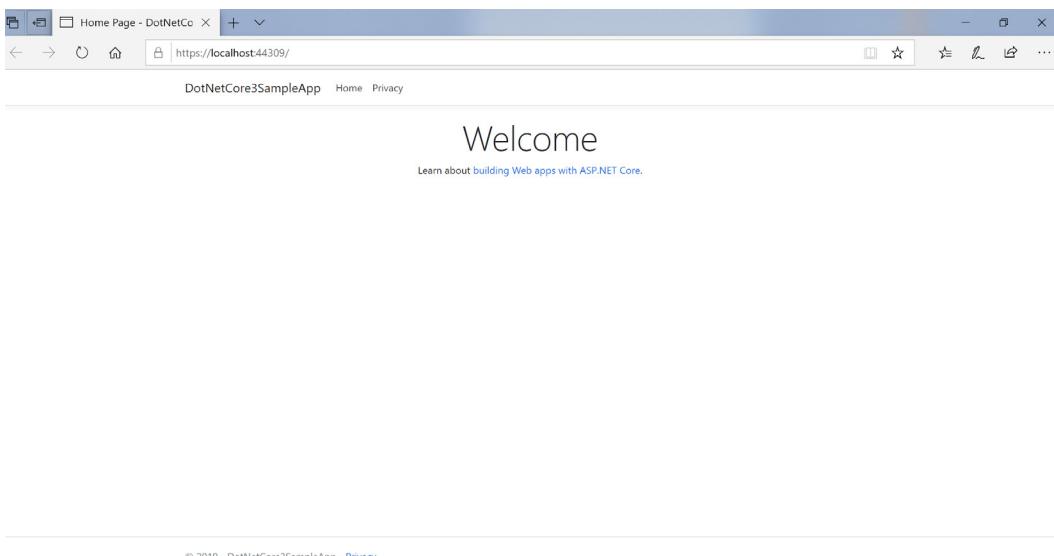


Figure 1.28: Running the .NET Core application

Now we can modify this app based on our project requirements. The basic structure, folders, and packages are created with the template. Through the book, we would create numerous applications while learning parallel programming.

Summary

In this chapter, we downloaded and installed the required tools and frameworks for leveraging .NET Core 3.1. We now have all the required tools and frameworks in our machine to start coding and experimenting and begin our journey of parallel programming with .NET Core 3.1 and C# 8. We created our first .NET Core 3.1 application using the template provided by Visual Studio. In the next chapter, we will discuss the .NET Core 3.1 framework, what is new in it and what it has in store for us.

Exercise

1. What is the difference between Visual Studio Enterprise and Visual Studio Code?
2. Can we use Visual Studio Professional with a Linux machine?
3. Why do we use JustDecompile?
4. What is the use of the ProcMon tool?
5. What is the use of WinDbg and DebugDiag?
6. Which tools can be used for dump analysis?

CHAPTER 2

What's New in C# 8?

"Change is the only constant in life."

C# is an advanced language and has excellent language features to build enterprise-grade applications. The beautiful thing about C# is that Microsoft keeps releasing feature updates periodically. In this chapter, we will discuss enhancements and new features that are shipped with C# 8.

Structure

We will discuss the following topics:

- C# 8 platform dependencies
- Nullable reference types
- Asynchronous streams
- Ranges and indices
- Default implementations of interface members
- Read-only members
- Pattern matching enhancements:
 - Recursive patterns
 - Switch expressions

- Property patterns
- Tuple patterns
- Using declarations
- Static local functions
- Disposable ref structs
- Null-coalescing assignment
- Interpolated verbatim strings enhancement
- Exercise

Objective

By the end of this chapter, the reader should:

- Know and list C# 8 platform dependencies
- Know the new features and enhancements shipped as part of C# 8
- Be able to understand and explain the practical implementation of each feature

C# 8 platform dependencies

Before starting our discussion on new features in C# 8, let's discuss C# 8 platform dependencies. Few of the new features shipped with C# 8 are dependent on the platform they are executed on. We will see these features a little later in the chapter, but features like Asynchronous streams, ranges, and indices rely on the new framework types that are part of .NET Standard 2.1. We are discussing .NET Core 3.1 in this book, which implements .NET Standard 2.1. The latest full .NET framework at the time of writing this chapter is .NET 4.8. .NET 4.8 does not implement .NET Standard 2.1, so the types essential to use these features are not available on .NET 4.8 full framework.

Xamarin, Mono, Unity, .NET Core 3.1 implements .NET Standard 2.1. Also, the default implementations of interface members depend on new runtime improvements, and not available in the .NET Runtime 4.8. So, in short, C# 8.0 full features are only available on platforms that implement .NET Standard 2.1. It is an important thing to keep in mind.

Now that we know about platform dependencies of a few of the features, we are now set to explore the new features and enhancements in C# 8.

New features and enhancements

At the time of authoring this book, (October 2019), the latest version of C# is 8.0. Many exciting features have been added in this new release. Let's have a look at these new features.

We will jump into details of each feature one by one.

Nullable reference types/Non-nullable reference type

One of the most frequently encountered exceptions in the .NET world is the null reference exception. We all may have seen these exceptions numerous times during our development, and I have even seen these issues in production environments. These need to be appropriately handled, else may lead to business impact or customer dissatisfaction.

Null reference exception could arise due to many cases; few possible scenarios are:

- **Missing input validation:** This is a typical user input scenario, where the application expects data input from the user. Now, data needs to be as per what the application expects. If the user enters incorrect input and the developer has missed appropriate validations to check if the user has entered valid data, it may result in a null reference exception.
- **Non-mandatory inputs:** There are cases when certain information is not mandatory to be entered but incorrectly used or expected later in the flow, and it results in null reference exception as the input wasn't provided.
- **Data returned from API or DB:** At times, the data returned from a service endpoint or database is processed, and if that data is null, it may result in a null reference exception.

These exceptions occur at run time and are caught only when they cause the damage. Few of these may be caused only occasionally as an edge case and may be hard to reproduce, so there was a need to have better compiler support to know about these while writing the code or in design time. Let us see what is new in C# 8. By default, all classes are reference types and hence nullable. The string is defined as a class in the framework and hence is a reference type and, therefore, nullable. With C# 8, this feature is made explicit; that is, reference types can be nullable as well as non-nullable, and compiler provides warnings knowing that you have expressed the intended purpose of the reference type.

In C# reference types can be categorized based on its usage as:

1. **Reference can null:** Variable can be initialized / assigned with a null value.

2. **Reference is not intended to be null:** In this case, the compiler enforces rules without checking value is null or not, to keep the reference not null. The reference variable must be initialized with a non-null value, and we cannot assign null to that variable.

In both cases, the declaration was the same in previous versions of C#, and no warning was raised by the compiler at the time of compilation. In such cases, we get a runtime error if wrongly null is assigned to a variable. But with C# 8, we can explicitly and clearly define the reference type variable as nullable or non-nullable.

Consider the following code snippet:

```
string str = null; // throws a warning: Assignment of null to non-nullable reference type
```

There is a null-forgiving operator "!" the variable name followed by "!. It overrides the compiler's analysis and removes warning. We can use this when we are sure that the variable cannot be null. For example, if we are sure that variable str is not null and we can find the str length, but compiler throws a warning, we can write the following code to override the compiler's analysis:

```
str!.Length;
```

In the above code, str is non-nullable because we have not declared it nullable using the ? (question mark). If you want to assign a null value to it, it should be marked as nullable using the ? as shown below, where the warning is gone:

```
string? str1 = null; // works fine
```

Note: Using the above expression, we can assign null, but if we use nullable reference, we need to apply the null check.

For example, in the below method, PrintLength takes a string as input, which could be null. Without a null check, it will throw null reference exception so we should write the null check as the first statement:

```
public void PrintLength(string? str)
{
    Console.WriteLine(str.Length); // Null reference exception
if str is null
}
```

The better solution would be to perform a null check before doing any operation on given input:

```
public void PrintLengthNew(string? str)
{
```

```
if (str != null)
{
    // Null check so we won't get here if str is null
    Console.WriteLine(str.Length);
}
```

The benefit of this null reference feature is, we can identify values that possibly could return null. So, handling it in code in advance with a null check and telling compiler in advance that this reference type could be null instead of knowing with the exception at runtime.

If we upgrade an existing .NET Core project to use .NET Core 3.1 from any other previous version then to enable nullable reference type for the entire project, we can just edit the project file and add a new property named **Nullable** to the property group and set it to enable:

```
<Nullable>enable</Nullable>
```

The compiler will now apply nullable reference types rules across the entire project and treat the code accordingly. This property and behavior are added by default for a new C# 8, .NET Core 3.1 projects, so we don't need to add any new property in newly created .NET Core 3.1 projects. We can also use directives to set contexts at any place in the project:

- To disable nullable reference warning and annotation context set: `#nullable disable`
- To enable nullable reference warning and annotation context set: `#nullable enable`
- To enable nullable reference warning set: `#nullable enable warnings`
- To disable nullable reference warning set: `#nullable disable warnings`
- To enable nullable reference annotation context set: `#nullable enable annotations`
- To disable nullable reference annotation context set: `#nullable disable annotations`
- To restore nullable reference warning set: `#nullable restore warnings`
- To restore nullable reference annotation context set: `#nullable restore annotations`

Asynchronous streams

Async methods are immensely popular and useful as it runs operations asynchronously and does not block the UI. Async methods return value asynchronously, but developers find one drawback that with `async await`, that we can return only one value. For example, we can return `Task<int>`, not multiple values. It does not return `IEnumerable<int>` types and cannot use `yield` keyword. We must wait to get a full dataset and then process it.

C# 8 comes with an enhancement on `async await`; now, we can yield return multiple values or sequence of values asynchronously. Also `await` is now can be used with a `foreach` loop. It supports lazy enumeration (`yield`) return with the `async` method.

New C# version added two new interfaces `IAsyncEnumerable<T>` and `IAsyncEnumerator<T>`, this is similar to `IEnumerable<T>` and `IEnumerator<T>`.

This enhancement is especially useful for processing data asynchronously, which is getting published by any publisher or read from the database. So now, for reading data based on its availability is possible by using `async await`. For example, in push-type communication, data will be displayed when a new message is pushed. We created a static list of strings `voters`. The `async` task `VoterNamesAsync` prints the name of voters as it comes from `voterListAsync`. We have used `await` keyword with `foreach` here and result displayed in the following output image:

```
static List<string> voters = new List<string>() { "Neha", "Rishabh",  
"Rahul", "Amit", "Juhi", "Namita", "Pallavi" };  
  
public static async Task VoterNamesAsync()  
{  
    await foreach(string voterName in  
voterListAsync(voters))  
    {  
        Console.WriteLine($"Next voter name is {voterName}  
and time is {DateTime.Now:hh:mm:ss}");  
    }  
}  
  
private static async IAsyncEnumerable<string>  
voterListAsync(List<string> votersname)  
{  
    int count = votersname.Count();  
    for (int i = 0; i < count; i++)  
    {
```

```

        if(votersname[i].StartsWith('R'))
        {
            await Task.Delay(2000);
        }

        yield return votersname[i];
    }
}

```

In *Figure 2.1*, if the name starts with 'R,' we add a delay of 5ms, and for the rest of the names, the result comes without any delay:

```

C:\Users\neshriv\Source\Repos\NewFeaturesOfCSharp8\bin\Debug\netcoreapp3.0\NewFeaturesOfCSharp8.exe
13
Next voter name is Neha and time is 10:27:50 → 5 ms delay on next
Next voter name is Rishabh and time is 10:27:55 → name , as logic says -
Next voter name is Rahul and time is 10:28:00 → delay by 5ms if next
Next voter name is Amit and time is 10:28:00 → name starts with R
Next voter name is Juhi and time is 10:28:00 → Running without
Next voter name is Namita and time is 10:28:00 → delay as name
Next voter name is Pallavi and time is 10:28:00 → doesn't start with 'R'

```

Figure 2.1: Output of VoterNamesAsync

The asynchronous method can be called on a need basis to return multiple values until it reaches the end of the enumerator. We will see them in action in *Chapter 5*.

Ranges and indices

As a developer/student, you might have heard a problem that "Write a program to find the second last value of an array" or "Create a subarray from input array with last five values." You iterate through the list to find out the last or second last element of the array list. Now you just need a single line of code. Operator ^ will do that for you and ranges .. will help.

C# 8 introduced two new operators and two new types to get the subrange from an array.

Note: An important point is the index from starting is counted from 0 to Length-1, but from the end, the last index would be the length of an array. So, to get the last index of an array writes 1 instead of 0 ; that's because the reverse index of an array is relative to the length of the sequence.

System.Index

Using this, we can get the specified index value from an array. Index type comes with an operator $^$. This operator gives the value of the index from the end of the array. Example:

Let's say we have a method named `FindAndPrintRange`, which prints the result. We have an array `arr` with integer values, and we want to print the second-last value of an array.

To print values from the end, we will use $^$ operator and write `arr[2]` for printing the second value from the end of the array list:

```
public static void FindAndPrintRange()
{
    int[] arr = new int[10] { 2,6,4,8,5,0,6,7,3,9};
    Console.WriteLine($"The second value from the end in array arr is {arr[ $^2$ ]}" );
    Console.WriteLine($"The last word in array arrStr is {arrStr[ $^1$ ]}" );
    Index i1 = 5; // fifth value from the starting
    Index i2 =  $^1$ ; // last value of an array or first value from the end of an array
    Console.WriteLine(arr[i1]); //0
    Console.WriteLine(arr[i2]); // 9
    Console.ReadKey();
}
```

System.Range

It represents a subrange of a sequence. The range operator is `(..)`, with start and end range using this `(...)` operator we can get the subrange. Example:

```
public static void FindAndPrintRangeStrings()
{
    var arrStr = new string[] { "Hello", "Friends", "Welcome",
    "To", "The", "Course" };
```

```
Console.WriteLine($"The last word in array arrStr is  
{arrStr[^1]}");  
  
Index i1 = 5;// fifth value from the starting  
Index i2 = ^1; // last value of an array or first value from  
the end of an array  
  
var subArrayOfWords = arrStr[2..5]; // it will return -  
"Welcome", "To", "The" ,last will not be included  
  
var subArr = arrStr[^6..^4]; // it will return - Hello  
,Friends ,  
  
var fullArr = arrStr[..]; // returns all values of an array  
  
foreach (string s in subArrayOfWords)  
{  
    Console.Write($"{s} ,");  
}  
Console.WriteLine();  
foreach (string s in subArr)  
{  
    Console.Write($"{s} ,");  
}  
Console.WriteLine();  
foreach (string s in fullArr)  
{  
    Console.Write($"{s} ,");  
}  
Console.WriteLine();  
Console.ReadKey();  
}
```

Let's see another example with integer values for more understanding:

```
public static void FindAndPrintRange()  
{  
  
    int[] arr = new int[10] { 2, 6, 4, 8, 5, 0, 6, 7, 3, 9 };
```

```
Console.WriteLine($"The second value from the end in array  
arr is {arr[^2]}");  
  
Index i1 = 5;// fifth value from the starting  
Index i2 = ^1; // last value of an array or first value from  
the end of an array  
Console.WriteLine(arr[i1]); //0  
Console.WriteLine(arr[i2]); // 9  
Console.WriteLine();  
var a1 = arr[3..]; // it will return all values starting  
from 3rd index 8,5,0,6,7,3,9  
  
var a2 = arr[..7]; // it will return all values till 7th  
index - 2,6,4,8,5,0,6  
  
var a3 = arr[3..7]; // 8,5,0,6  
  
foreach (int i in a1)  
{  
    Console.Write($"{i} ,");  
}  
Console.WriteLine();  
foreach (int i in a2)  
{  
    Console.Write($"{i} ,");  
}  
Console.WriteLine();  
foreach (int i in a3)  
{  
    Console.Write($"{i} ,");  
}  
Console.ReadKey();  
}
```

In the following *Figure 2.2* output of the above program to find last index value and range from an array:

```
C:\Users\neshriv\Source\Repos\NewFeaturesOfCSharp8\bin\Debug\netcoreapp3.0\NewFeaturesOfCSharp8.exe
The second value from the end in array arr is 3
0 → Fifth value from the starting - { 2,6,4,8,5,0,6,7,3,9}
9 → Last value of array - { 2,6,4,8,5,0,6,7,3,9}

8 ,5 ,0 ,6 ,7 ,3 ,9 , → returns all values starting from 3rd index
2 ,6 ,4 ,8 ,5 ,0 ,6 , → returns all values till 7th index
The last word in array arrStr is Course
Welcome ,To ,The , → returns 2nd index till 5th
Hello ,Friends , → return 6th from the last till 4th from the last
Hello ,Friends ,Welcome ,To ,The ,Course , → returns all values of an array
```

Figure 2.2: Output of `FindAndPrintRange` and `FindAndPrintRangeStrings`

Default implementations of interface members

How many times have you thought of adding a new method in an interface and stopped! Thinking that you must analyze first how many places we have to add this method to avoid breaking the existing code where that interface is used.

You might think that what if we don't have to make changes in all the places and, at the same time, able to add a new method in the interface. C# 8 is here to solve this problem! C# 8 provides a default implementation for methods in the interface, so no need to worry about breaking existing code as it will just take care of it by using default implementation. Isn't it a cool feature! Let's jump into the code to see how we can do this:

```
interface IAccount
{
    void Credit(int amount, string message);
    void Debit(int amount, string message);
    // New overload
    void Credit(int amount) => Console.WriteLine($"{amount} is
credited in your account");
}

class UserAccount : IAccount
{
    public void Credit(int amt, string message) { Console.
```

```
WriteLine($"{message} : {amt}"); }  
public void Debit(int amount, string message) { Console.  
WriteLine($"{message} : {amount}"); }  
// void Credit(int amount) gets it's default implementation  
}
```

In *Figure 2.3*, we are using the default method of the interface which is not implemented in `UserAccount` class:

```
IAccount userAccount = new UserAccount();  
userAccount.Credit(90000);
```

Figure 2.3: Using default method of interface

Here we created an interface `IAccount` which contains method `Credit` and `Debit` with parameter amount and message. The amount which is credited or debited with a message given in the message parameter.

`UserAccount` is a class that implemented the `IAccount` interface.

Let's add another `Credit` method in interface `IAccount`, which doesn't have any message, and in case of `Credit` without a message, we added a default message with the amount.

In C# 8, we can add new methods in the interface without worrying about breaking the code at all the places where that interface is referred. We can provide a default implementation in the interface. Because of default implementation, it will automatically be referred to in all the places where the interface is being used, and nothing will break.

`UserAccount` class gets `Credit(int amount)`'s default implementation, so `UserAccount` doesn't have to implement this `Credit` method.

So, using C# 8, we can add any number of methods with default implantation in the interface, which is referred to at multiple places without breaking existing implementers as they will get default implementation:

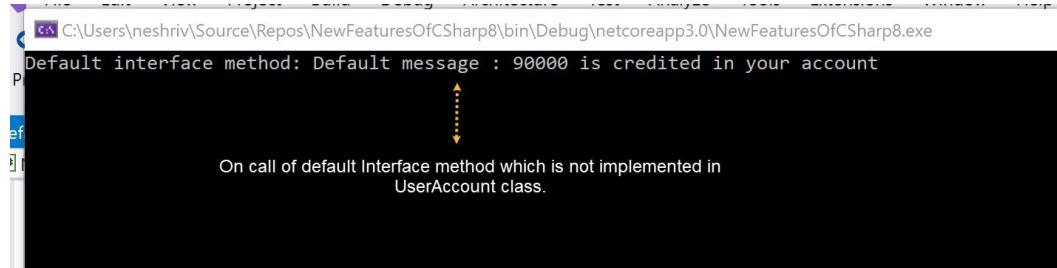


Figure 2.4: Output of default interface method

Readonly members on structs

`Readonly` is a minor feature addition on new C# version 8. Instead of applying a `readonly` modifier at the `struct` level, we can apply a `readonly` modifier to any member of a `struct`, which guarantees that its state will not change.

Suppose I have a `struct CityDistance` and we are setting `TwoCityDistance`, which is not `readonly` and depends on properties `A` and `B`. `ToString()` is `readonly` and consuming `TwoCityDistance` which is not a read-only property. If a member is using a property that can be changed and not `readonly`, then the compiler will throw a warning, and for the safer side, it creates an implicit copy of it. Please see the following screenshot:

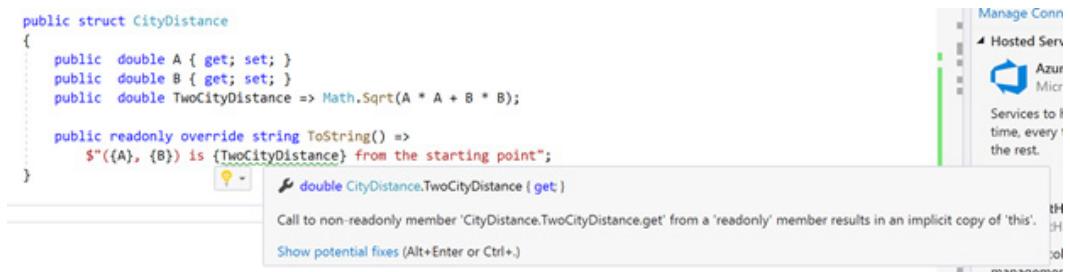


Figure 2.5: Warning of using a non-readonly variable is read-only member

Figure 2.5 showing warning of using a non-`readonly` variable in `readonly` member as `readonly` is not declared at `struct` level. We are marking `readonly` to members of a `struct`, which is needed. To improve this, we can make `TwoCityDistance` as `readonly`. Doing this will remove the warning, and also, we don't have to mark `struct` as `readonly`.

Pattern matching enhancements

C# 8 added new features to pattern matching, which was announced with C#7. It also includes switch case pattern enhancement.

New patterns introduced as part of C# 8 is recursive pattern and property pattern. Let's first see the enhancement of switch expression and then recursive pattern followed by property pattern with examples.

Switch expressions

With new switch expression, we need not write repetitive keywords like "case:" and `break`, the following keywords are no longer needed for switch statement:

- **case::** is replaced by Lambda `=>`
- **default:** It is replaced by `_`

- **switch:** Keyword would be infix between test value and the test cases like:

```
public enum Cities
{
    Delhi,
    Mumbai,
    Chennai,
    Hyderabad,
    Bangalore
}
```

Old switch statement:

```
public static bool IsCityHasElectionold(Cities city)
{
    switch (city)
    {
        case Cities.Delhi:
            return true;
        case Cities.Mumbai:
            return true;
        case Cities.Chennai:
            return true;
        case Cities.Hyderabad:
            return true;
        case Cities.Bangalore:
            return true;
        default:
            return false;
    };
}
```

New switch statement:

```
private static string CityInState(string city)
{
    if (city == "Delhi") return "Delhi";
    if (city == "Mumbai") return "Mumbai";
    if (city == "Chennai") return "Chennai";
    if (city == "Hyderabad") return "Hyderabad";
    if (city == "Bangalore") return "Bangalore";
}
```

```
        else if (city == "Mumbai") return "Maharashtra";
        else if (city == "Chennai") return "Tamilnadu";
        else if (city == "Hyderabad") return "Telangana";
        else if (city == "Bangalore") return "Karnataka";
        else return "NA";
    }

    public static bool IsStateHasElection(States states)
    {
        bool hasElection = states switch
        {
            States.Delhi => true,
            States.Maharashtra => false,
            States.Tamilnadu => true,
            States.Telangana => false,
            States.Karnataka => true,
            _ => false
        };
        return hasElection;
    }

    public static string IsCityHasElection(string city, States
states) =>
        (CityInState(city), IsStateHasElection(states)) switch
    {
        ("Delhi", true) => "Delhi has election",
        ("Maharashtra", false) => "Maharashtra doesn't have
election",
        ("Tamilnadu", true) => "Tamilnadu has election",
        ("Telangana", false) => "Telangana doesn't have
election",
        ("Karnataka", true) => "Karnataka has election",
        (_,_) => "Wrong input",
    };
}
```

If we compare both switch expressions written above, we can notice that the newer one is easy to write with fewer keywords. Also, with the new `switch` statement, we can create a `tuple` with the values we want to check. Here we are concerned about `City` belongs to which state and that state has an election or not. Based on the result it returns, we check the below conditions and return a message.

In new switch expression, we keep the variable first and then `switch` keyword. We can set the variable value as a result of switch expression and then return. For example, in method `IsStateHasElection` has a bool variable `hasElection`, value is assigned to this variable based on switch statements and returned. We can also return the result directly, and we can convert the method to an expression body, as shown in the second example `IsCityHasElection`, which is returning string value as a result of the expression.

New switch expression increased the readability of code and came with more flexible options. We can use patterns with switch statements like property patterns, recursive patterns, tuple patterns.

Recursive patterns

Any pattern expression results in expression, and as the name suggests, the recursive pattern is a pattern where one pattern expression applied to another pattern expression. In simple words, patterns to contain other patterns are allowed.

It's a fantastic feature, gives you the adaptability to test information against a succession of conditions and perform further calculations dependent on the conditions met:

```
class ElectionCity
{
    public string Name { get; set; }
    public bool HasElection { get; set; }
    public string State { get; set; }
    public ElectionCity(string name, bool election, string state)
    {
        this.Name = name;
        this.State = state;
        this.HasElection = election;
    }
}
ElectionCity EC1 = new ElectionCity("Delhi", true, "Delhi");
```

```

ElectionCity EC2 = new ElectionCity("Hyderabad", true,
"Telangana");
ElectionCity EC3 = new ElectionCity("Chennai", false,
"Tamilnadu");
public List<ElectionCity> cities = new List<ElectionCity>() {};
IEnumerable<string> GetCityNames()
{
    foreach (var city in cities)
    {
        if (city is { HasElection: true, Name: string name })
yield return name;
    }
}

```

Here we have created a class `ElectionCity`, which has three properties—`Name`, `HasElection`, `State`. It has a public constructor that assigns these values.

Inside if condition pattern `{HasElection: true, Name: string name}` verifies that this city has election or not. If `HasElection` is true and the city name is not null, it returns the city name, which has an election.

The recursive pattern also consists of sub-patterns—Positional pattern and Property pattern.

Positional pattern

The positional pattern is a type of recursive pattern, so it contains nested patterns. It can be used to identify that tuple meets the criteria. We can use this with switch statements. To know more about patterns, go to link <https://devblogs.microsoft.com/dotnet/do-more-with-patterns-in-c-8-0/>.

Property pattern

The property pattern empowers you to coordinate on the properties of the item analyzed. Let's take the same example of cities where the election will happen. Depending on the name of the city, we can find out where the election will be conducted. Yet several voting booths differ dependent on the city's population. That calculation isn't a primary duty of a `City` class. `City` class consists of three properties `City name`, `it's population` and whether this city belongs to India or not:

```

class CityDetail
{

```

```
public string Name { get; set; }

public long Population { get; set; }

public bool IsInIndia { get; set; }

public CityDetail(string name, long population, bool isInIndia)
{
    this.Name = name;
    this.Population = population;
    this.IsInIndia = isInIndia;
}

}
```

The number of voting booths relies upon the city's population, and we only calculate it if the city belongs to India. The `CalculateBoothCount` method computes the required number of booths in a city based on its population using property pattern:

```
class PropertyPattern
{
    CityDetail Hyderabad = new CityDetail("Hyderabad", 150000, true);

    public static int CalculateBoothCount(CityDetail city, int
numberOfBooths = 1) =>
        city switch
    {
        { Population: 10000, IsInIndia:true } => numberOfBooths + (10000 /
200),
        { Population: 150000, IsInIndia: true } => numberOfBooths +
(10000 / 500),
        { Population: 200000, IsInIndia: true } => numberOfBooths +
(10000 / 700),
        _ => 1
    };
}
```

With the `switch` statement, we used `PropertyPattern`. In the above code, we are returning several booths as an integer value. Using `PropertyPattern`, we can apply multiple checks on different properties such as to calculate the number of booths, `{prop1,prop2,...}` (example: `{Population: 10000, IsInIndia:true}`) we are checking the population of city and city belongs to India, if both conditions are right, calculating the booth count. Using `{..}` and comma-separated property values to check.

Note: For property pattern and rest, all type patterns value should not be null. So, there could be a case when we have an empty property; in that case, we can pass a not null object. For example: `{ } => obj.ToString();` and we can set `null => "null";`

Tuple patterns

Tuple patterns allow matching of more than one value (a tuple) in a `switch` expression. A `switch` statement can be applied to a tuple. Tuple allows us to select a case based on multiple criteria, which can be passed as a `tuple`. For example, eligibility for voting in India includes age should be more than 18 years, and nationality should be Indian. These two conditions, age and nationality, we can put in a tuple and decide voting right. We created an enum `Nationality`, which contains country names and the `switch` statement is applied on a `tuple(age, nationality)`, and it returns string result:

```
public enum Nationality
{
    Indian,
    USA,
    Canadian,
    SA,
    UK,
    China
}

class TuplePatterns
{
    public static string RightToVote(int age, Nationality
nationality)
    => (age, nationality) switch
    {
```

```
(6, Nationality.Indian) => "No you can't vote for now",
(17, Nationality.USA) => "No you can't vote in India",
(20, Nationality.UK) => "No you can't vote in India",
(33, Nationality.Indian) => "Yes! Go ahead and vote for India",
(67, Nationality.Indian) => "Yes! Go ahead and vote for India",
(_ , _) => "Can't say!"

};

}
```

In the preceding example, we are returning a message based on two conditions. Similarly, in the tuple, we can pass multiple criteria for a case to pass.

Using declarations

When we use `using` keyword, we tell the compiler to dispose of the variable which is declared in `using` at the end of the scope. The end of the scope is declared by putting parenthesis of start and close for `using`. If we need to use multiple `using` statements, in that case, readability of code decreases, and track the opening and closing of each parenthesis becomes a pain.

In C#8, we don't need to keep track of nested parentheses to keep the scope of variables declared in `using` statements. Variable will be disposed of at the end of its scope.

For example, previously, we used to write like below for defining the scope of the variable. Here we declared a `datafile` variable `StreamWriter`, which writes data in `ResultData.txt`, inside a `using` statement.

There could be possibly more than one `using` statements, which would be nested and datafile variable get disposed when the closing bracket associated with the `using` statement is reached:

```
static void WriteToFile(IEnumerable<string> textLine)
{
    using (var datafile = new System.IO.StreamWriter("ResultData.
txt"))
    {
        foreach (string currentline in textLine)
        {
            if (currentline.Length != 0)
            {
```

```

        datafile.WriteLine(currentline);
    }
}
// datafile will be disposed here
}
}

```

In C#8, we don't need to keep track of parenthesis of using statements and when the variable is getting disposed of. It gets disposed of when the closing bracket of the method reaches. If multiple using statements are used, all variables get disposed of once the closing parenthesis of the method reaches.

In both cases, the compiler makes the call to `Dispose()`. The compiler creates an error if the expression in the using statement is not disposable:

```

static void WriteToFileNew(IEnumerable<string> textLine)
{
    using var datafile = new System.IO.StreamWriter("ResultData.
txt");
    foreach (string currentline in textLine)
    {
        if (currentline.Length != 0)
        {
            datafile.WriteLine(currentline);
        }
    }
} // datafile will be disposed of here

```

Static local functions

"Static" local function is introduced with C# 8; before this, C#7 came up with the idea of local functions that can be used with `async` and `unsafe` modifiers. Allowing static modifiers enhanced as part of C#8.

Local functions are methods declared/defined inside another method; it can be nested. It gives us a better understanding of the method's context and limitations where it can be used.

Local functions automatically take the context of the method inside which it is written, to create any variables from the containing function available inside them.

Using static with local function safeguard that the local function doesn't refer to any variables from the enclosing scope or outer scope. Let's try to understand this by example:

```
Console.WriteLine("Enter population of city, to know number of voting
booths required!");
long population = Convert.ToInt64(Console.ReadLine());
int numberOfBooths = NumberOfBooths(population);
Console.WriteLine(numberOfBooths);
public int NumberOfBooths(long population)
{
    int votingBoothCount;
    CalculateBoothCount(population);
    return votingBoothCount;
    // non-static local function which is using variable of main
    function and setting variable value of it.
    void CalculateBoothCount(long population)
    {
        votingBoothCount = Convert.ToInt32(population / 500);
    }
}
```

Here we are reading the population from the console and returning the number of voting booths needed for that population.

Function `NumberOfBooths` is taking the population as an input parameter, and we have created a `votingBoothCount` variable inside this method. `CalculateBoothCount` is a local function that calculates the count of voting booths and assigns it to variable `votingBoothCount`, which we return from the method - `NumberOfBooths`.

Let's see the following screenshot where I added `static` in the local function. Now because we are trying to make local function as `static`, it is throwing an error that

`static` local function cannot contain a reference to `votingBoothCount`. The `static` local function cannot access any variable of enclosing function:

```

public int NumberOfBooths1(long population)
{
    int votingBoothCount;
    CalculateBoothCount(population);
    return votingBoothCount;
    // non-static local function which is using variable of main function and setting variable value of it.
    static void CalculateBoothCount(long population)
    {
        votingBoothCount = Convert.ToInt32(population / 500);
    }
}

```

A static local function cannot contain a reference to 'votingBoothCount'.

Figure 2.6: Error on using a variable of mail function in a static local function

In Figure 2.6 error saying that we cannot use a variable in the scope of the method, which defines a static local function. The static local function can only use a variable in the local static function's scope.

So, how we can write a static local function is explained in below code snippet:

```

public int NumberOfBoothsUsingStatic(long population)
{
    int votingBoothCount;
    votingBoothCount = CalculateBoothCount(population);
    return votingBoothCount;

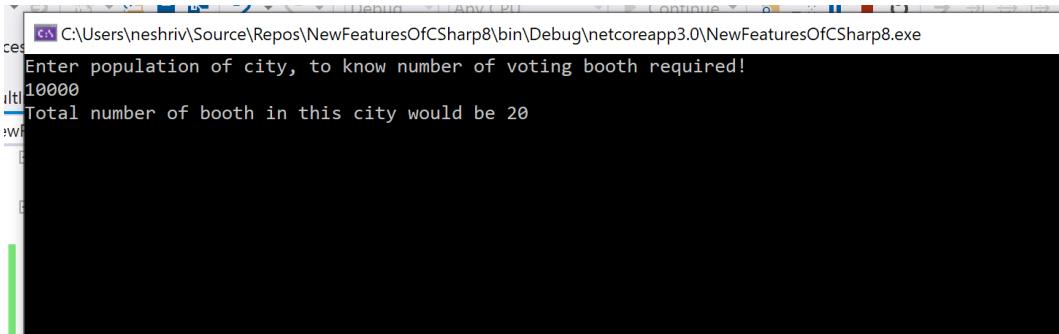
    //static local function
    static int CalculateBoothCount(long population)
    {
        return Convert.ToInt32(population / 500);
    }
}

```

In the above code, we created a static local function `CalculateBoothCount`, which returns a value. in the main body of method `NumberOfBoothsUsingStatic`, we are assigning the returned value of the local static function to variable `votingBoothCount`.

In the preceding example, the static local function is not referring to any variable which is outside the scope of the local function.

Output:



```
C:\Users\neshriv\Source\Repos\NewFeaturesOfCSharp8\bin\Debug\netcoreapp3.0\NewFeaturesOfCSharp8.exe
Enter population of city, to know number of voting booth required!
10000
Total number of booth in this city would be 20
```

Figure 2.7: Output of program using local static function

Disposable ref structs

Cleanup is one of the most critical and discussed topics. Deterministic is more preferred over the not deterministic finalize. The best practice is to explicitly use the `Dispose` method or `Using` statement when an object is no longer needed instead of waiting for it to be cleared by the execution of runtime finalizer.

C# provides an `IDisposable` interface which implements the `Dispose` method. Still in case of `ref struct` which introduced as part of C# 7.2, we cannot implement the interface, and without implementing `IDisposable`, we cannot use the `Dispose` method, and hence we can't use `ref struct` in using statement:

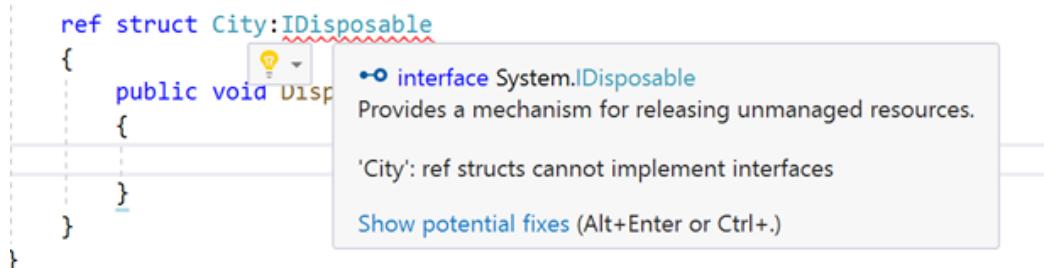


Figure 2.8: Error on using `IDisposable` with `ref struct`

Figure 2.8 shows an error in implementing `IDisposable` with `ref struct`. We cannot implement an interface with `ref struct`.

C# 8 comes with the solution to this problem. Now we can write the `public Dispose` method in `ref struct`, and using statement takes it up. Let's see the code:

```
ref struct City
{
    public void Dispose()
    {
    }
}
```

Figure 2.9: Shows solution of problem specified in Figure 2.8

We can directly use the `Dispose` method without using `IDisposable` in `ref struct`:

```
static void Main(string[] args)
{
    using (var city = new City())
    {
        Console.WriteLine("Hello Hyderabad!");
    }
}

ref struct City
{
    public void Dispose()
    {
    }
}
```

Null-coalescing assignment

The Null-coalescing assignment operator is a new assignment operator `??=` introduced with C# 8. This operator combines functionality in one, first checking that value is `null` or not, and second, if it is `null`, assign the value to the variable. In the following example method, `AddCitiesForElection` takes the city name as an input parameter, and we add the city's name to a list `lstcity`. We created a new string variable `newcity` and assigned `null` value to it. In the list `lstCity`, we added

a city named Raipur. If we now print it, we will get Raipur; now we are adding newCity to the list and assigning value Jaipur if it is null. In our case, newCity is null so, Jaipur will be assigned to variable newCity, and it will be added to the list. If we print the list now, we will get Raipur Jaipur result in the output window:

```
public static void AddCitiesForElection(string city)
{
    List<String> lstCity = new List<string>();
    string newCity = null;
    lstCity.Add("Raipur");
    Console.WriteLine(string.Join(" ", lstCity)); // returns
Raipur

    lstCity.Add(newCity ??= "Jaipur");
    Console.WriteLine(string.Join(" ", lstCity)); // output:
Raipur Jaipur

    Console.WriteLine(newCity); // output: Jaipur
}
```

Interpolated verbatim strings enhancement

Earlier we could only use @ after \$ symbol but now '\$@"...."' and '@\$"...."' both are allowed. Any order is valid.

\$ is a unique character used to find a string that needs to be interpolated. String variable value is replaced by its current value at the time of expression evaluation. For example:

```
string country = "India";
Console.WriteLine($"I am citizen of {country}, It's beautiful ! ");
@ is a special character used to prefix a code which compiler infer as an identifier or
string take as an interpreted verbatim. For example:
```

```
string folderLocation = @“ C:\Program Files\Microsoft”;
Console.WriteLine(folderLocation);
```

The above command will return C:\Program Files\Microsoft, instead of writing address using double slash \\ to read this address.

Summary

C# 8 comes up with enhancements and features which will change the way we code, more readable, flexible. We can declare invalid reference types, quickly find out range

or element from last of an array, also asynchronously get the list of items and use it. Using the new switch statement, we can verify more than one value and a more readable format. Let's check our understanding on this below Questions section.

Exercise

1. Can we assign `null` to value types?
2. Can we return multiple values in the `async` method?
3. How to restrict local function from using variables defined in a scope where the local function is defined.
4. Write a code to print 3rd value from the last in array - {1,2,3,4,5,6,7,8,9}
5. Write a code to print values from 2nd last till 4th from the end in an array - {1,2,3,4,5,6,7,8,9}
6. Can we add a method in an interface without affecting existing implementation at all pages where the interface is used? If yes, how to add the new method in the interface to avoid breaking at all pages.
7. Can we implement `IDisposable` with `ref struct`? How can we implement the `Dispose` method with `ref struct`?

CHAPTER 3

.NET Core 3.1

"If you want something new, you have to stop doing something old!"

~ Drucker

Introduction

.NET Core is Microsoft's great move towards cross-platform, Microsoft announced its last .NET Core version 3.1 in the year 2019. Next, .NET Core features will be merged with .NET full framework, and there will be one combined and highest version of .NET with the best of both worlds (.NET and .NET Core). This next version with combined functionality will be .NET 5 and is planned to release in November 2020. In November 2019, Microsoft released .NET Core 3.1 with Long term support, which will be supported for at least three years, and it will be a simple upgrade.

In this chapter, we are going to learn about what is new in .NET Core 3.1 and what all modifications are done from .NET Core 2.2 to .NET Core 3.1. .NET Core 3.1 comes with many features, but we will cover the main features like its most significant enhancement is Windows Desktop application support. We can now create Windows forms, WPF, and UWP in .NET Core. Also, the .NET Core 3.1 version comes with C# 8 support, as it supports .NET Standards 2.1, which we discussed in Chapter 2, under the platform dependencies section. We have used the template and created our first .NET Core 3.1 application in *chapter 1*. Now we will discuss all the enhancements in detail.

Please note that .NET Core 3.1 contains a set of bug fixes and refinements over .NET Core 3.0 with changes primarily focused on Blazor and Windows desktop, so in this book, if you see something targeted or explained on .NET Core 3.0, it applies equally to .NET Core 3.1

Structure

Topics to be covered are:

1. .NET Core APIs
2. Windows Desktop application support
 - o Windows Desktop Deployment MSIX
3. COM-callable components - Windows Desktop
4. WinForms high DPI
5. .NET Standard 2.1
6. C# 8 and its new features support
7. Compile and Deploy
 - o Default executable
 - o Single file executable
 - o Assembly linking
 - o Tiered compilation
 - o ReadyToRun images
 - o Cross-platform/architecture restrictions
8. Runtime/SDK
 - o Build copies dependencies
 - o Local tools
 - o Smaller Garbage Collection heap sizes
 - o Garbage Collection Large Page supports
 - o Opt-in feature
9. IEEE Floating-point improvements
10. Fast built-in JSON support
 - o Json Reader
 - o Json Writer
 - o Json Serializer
11. HTTP/2 support
12. Cryptographic Key Import/Exports

13. Summary

14. Exercise

Objective

After reading this chapter, the reader would:

- Learn about the new features added in .NET Core 3.1
- Be able to create a Windows desktop application using .NET Core 3.1
- Answer the quiz to test knowledge about .NET Core 3.1 new features

New features and enhancements

In this section, we will discuss improvements done on .NET Core APIs as part of the new version 3.1.

NET Core version APIs

As part of new enhancements, APIs versioning scheme has changed, which was used for getting the version detail. Now .NET Core 3.1 version API will return the familiar version name as a result. For example:

```
static void Main(string[] args)
{
    Console.WriteLine($"Environment.Version: {System.
Environment.Version}");

    Console.WriteLine();

    Console.WriteLine($"RuntimeInformation.FrameworkDescription:
{System.Runtime.InteropServices.RuntimeInformation.
FrameworkDescription}");

    Console.ReadKey();
}
```

Output:



```
C:\Users\neshriv\Source\Repos\NewFeaturesOfCSharp8\bin\Debug\netcoreapp3.0\NewFeaturesOfCSharp8.exe
Environment.Version: 3.0.0 → Environment Version: 3.0.0 instead of 4.0.30319.42000

RuntimeInformation.FrameworkDescription: .NET Core 3.0.0 → Runtime Info Framework Description: ".NET
Core 3.0.0" instead of ".NET Core
4.6.27415.71"
```

Figure 3.1: .NET Core 3.1 Version API result example

Windows Desktop application support

Using .NET Core 3.1, we can create a Windows desktop application, WPF application. Windows Form and WPF are integrated with .NET Core 3.1 build. Windows desktop component is a feature of Windows .NET Core 3.1 SDK. We can now use `dotnet` commands in .NET Core CLI to create a new Windows Form/WPF application. Following commands, we can use:

```
dotnet new winforms
```

```
dotnet new wpf
```

With Visual Studio 2019, the new templates are available for .NET Core 3.1 Windows applications. Once we click on create a new project; we can select a template from available templates for .NET Core 3.1 Windows application as shown in the following screenshot:

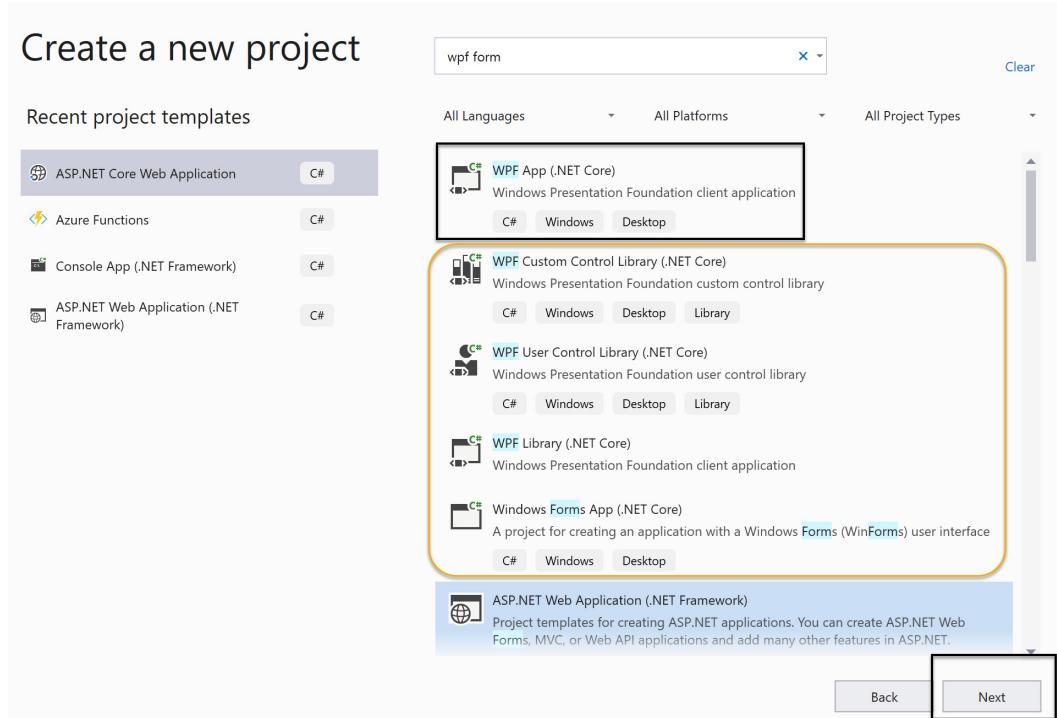


Figure 3.2: Create new project

We can see the available templates of .NET Core related to WPF and Windows Form. Out of these available templates, we can choose as per our requirement for WPF and Windows Form App with .NET Core, or we can create a new project and add references manually. For example, we have selected the WPF App (.NET Core)

template and then click the **Next** button. In the next step, provide a valid and suitable project name and project location and then click on the **Create** button.

For illustration, we will create a WPF application that is running on .NET Core and using WinUI features with XAML islands. For standard WinUI controls, Microsoft has a pre-built NuGet package that contains wrappers. We will add **Microsoft.Toolkit.Wpf.UI.Controls** package into our project. This NuGet package has dependencies, as shown in *Figure 3.3*, which also be loaded with this package:

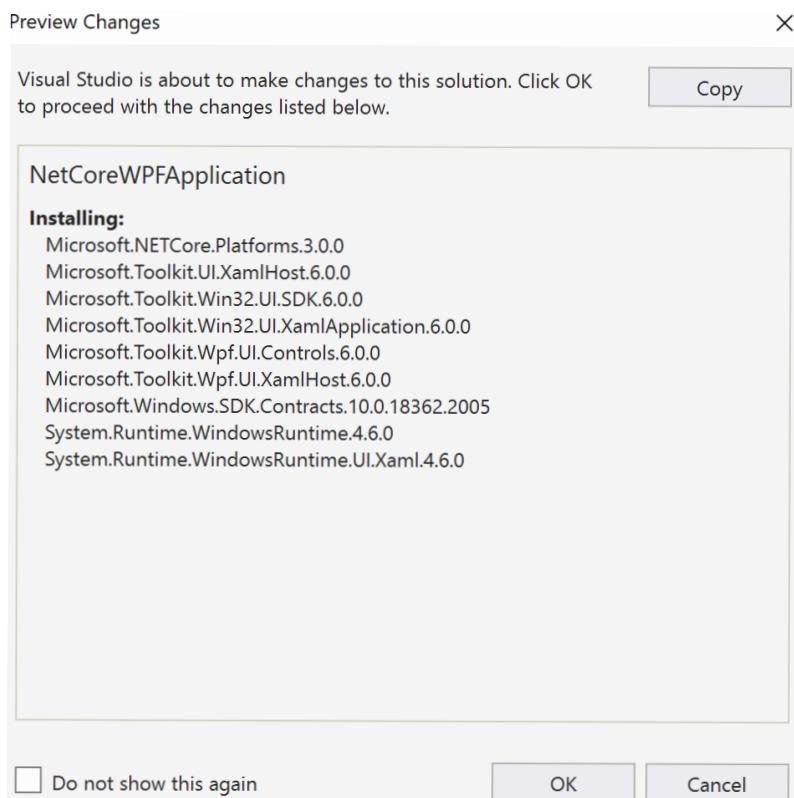


Figure 3.3: Microsoft.Toolkit.Wpf.UI.Controls package and its dependencies

After installing **Microsoft.Toolkit.Wpf.UI.Controls** NuGet package , we need to register the namespace **Microsoft.Toolkit.Wpf.UI.Controls** for our new controls in XAML file. So, add namespace in XAML as shown in the following code:

```
xmlns:Control = "clr-namespace:Microsoft.Toolkit.Wpf.  
UI.Controls;assembly=Microsoft.Toolkit.Wpf.UI.Controls"  
  
<Control:InkToolbar x:Name="toolbar" TargetInkCanvas="{x:Reference  
Nehacanvas}" Grid.Row="1" HorizontalAlignment="Left"  
VerticalAlignment="Top" Margin="5,5,5,5" Width="200" Height="60"> </  
Control:InkToolbar>
```

```
<Control:InkCanvas x:Name="Nehacanvas" Grid.Row="4"></
Control:InkCanvas>
```

Next, we added `InkToolbar` and `InkCanvas` control in XAML. In `xaml.cs`, we will define the allowed devices to write in canvas, which we added in XAML.

We must tell this in canvas to allow input from my mouse, so in code behind file `xaml.cs`, we will add namespace `Microsoft.Toolkit.Win32.UI.Controls.Interop.WinRT` and supported device type as a mouse:

```
using System.Windows.Shapes;
using Microsoft.Toolkit.Win32.UI.Controls.Interop.WinRT;
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        Nehacanvas.InkPresenter.InputDeviceTypes =
CoreInputDeviceTypes.Mouse;
    }
}
```

We can run our application now, and if it works, we should be able to draw with the mouse:

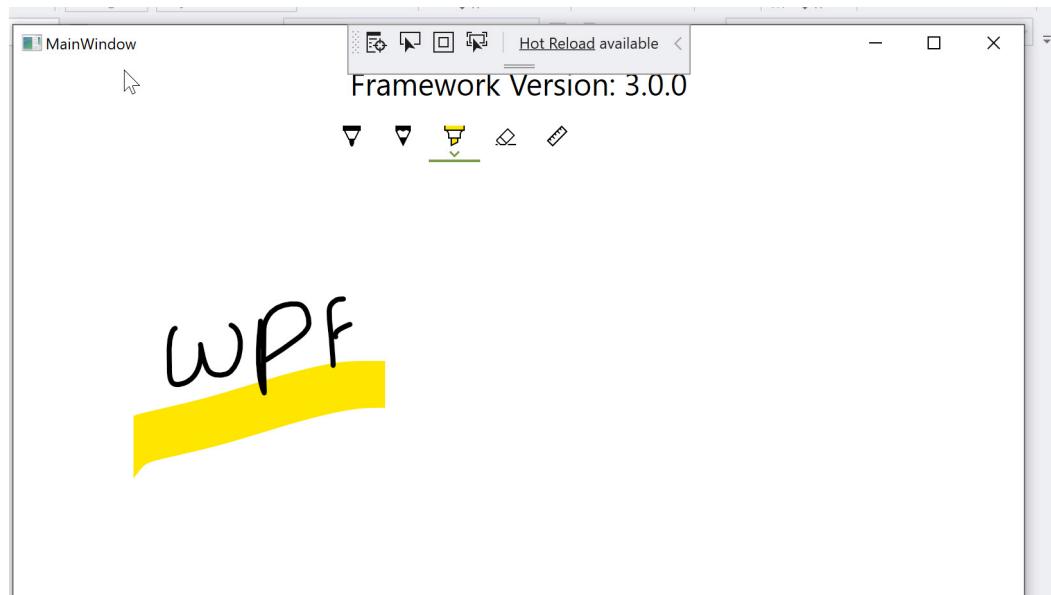


Figure 3.4: .NET Core 3.1 WPF application

Windows Desktop Deployment MSIX

Now we can build and deploy self-contained .NET Core applications using MSIX based packages. The Windows Application Packaging project, which can be installed with Visual Studio 2019 and can be used to create a self-contained package for your .NET Core apps. These packages contain our application dependencies, including .NET Core run time. We can then distribute this package by windows store or directly onto PCs.

COM-callable components – Windows Desktop

Com activation for .NET Core classes was essential for enabling existing .NET framework users to adopt .NET Core. Many users were not able to migrate to .NET Core because it was not supporting COM callable components.

We can create COM callable managed components on windows using .NET Core. To understand more about COM activation, please visit Microsoft document on COM-activation on GitHub. Go to the link: <https://github.com/dotnet/core-setup/blob/master/Documentation/design-docs/COM-activation.md>.

WinForms high DPI

DPI is **dots per inch**. To make a desktop application that should handle any display and change dynamically and remain clear and in excellent resolution, high DPI windows form application came into the picture. If we create a new Windows application, it is suggested to make a Universal window platform UWP app because it dynamically scales based on the display on which application is running. Still, older applications like win forms, WPF are unable to handle dots per inch DPI scaling dynamically.

In .NET Core Windows application, we can set high DPI mode:

```
public static bool SetHighDpiMode (System.Windows.Forms.HighDpiMode  
highDpiMode);
```

In this method, we specify the enum value of `highDpiMode`. Applying `highDpiMode` is dependent on the OS version on which the machine is running. Learn more about this from Microsoft documentation: https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.application.sethighdpmode?view=netcore-3.0#System_Windows_Forms_Application_SetHighDpiMode_System_Windows_Forms_HighDpiMode.

.NET Standard 2.1

Visual Studio 2019 supports .NET Core 3.1 and .NET Standard 2.1. With the new version of .NET Core 3.1, .NET Standard 2.1 is supported through default template is pointing to .NET Standard 2.0, we can later change it to `netstandard2.1` in the project file like shown in the following code:

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

    <PropertyGroup>
        <TargetFramework>netstandard2.1</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
    </ItemGroup>

</Project>
```

C# 8 and its new features support

As stated above that .NET Core 3.1 supports .NET Standard 2.1. We also discussed in *Chapter 2* that all new C# 8 features available in .Net Standard 2.1, and hence all new C# 8 features are available with .NET Core 3.1, like nullable reference type, switch statement, patterns and async stream which we discussed in *Chapter 2* (new features and enhancement of C#8). If .NET Core 3.1 is not supporting C#8 features, check the `TargetFramework` property in the project file and set it to `netstandard2.1`.

Compile and Deploy

Default executable

For .Net Core Apps, we can do the deployment in three ways:

1. **Framework Dependent Deployment (FDD):** This deployment depends on the existing version of .NET Core available on the target machine. In this deployment type, the deployment package contains only application code and DLLs, which can initiate using .net utility and third-party dependencies, which are not part of .NET Core.
2. **Self-Contained Deployment (SCD):** As the name suggests, this deployment doesn't depend on the version available on the target machine and isolated

from other .NET Core applications. In this type of deployment, the deployment package contains .NET Core libraries and runtime with application code.

3. **Framework Depended Executable (FDE):** This is introduced with .NET Core 2.2, where we can deploy our application with all its dependencies; it could be third party dependencies and based on the version installed on the target machine. These executables depend on .NET Core available on the target machine; it's not self-contained.

This **Framework Dependent Executable (FDE)** now defaults build with .NET Core 3.1. There are many benefits of FDE, few of them are:

- Deployment package size is smaller
- Improvement on disk usage as all .NET Core App utilizing same
- Net Core installation
- The application can be invoked by calling executable with no need of referring dotnet utility like a command: `dotnet example.dll`

Single executable file

.NET Core 3.1 enables applications to be published and distributed as a single executable file.

Aim of the single executable file is to make it broadly compatible for all applications, whether it has ready to run or MSIL assemblies, or it has config files, native libraries, and so on.

Benefits of the single executable file are:

- Integration with .NET CLI.
- Consistent experience for all applications.
- Third party tools may have used public APIs, which could be used in application as well, which may cause conflicts. This situation can be avoided by an inbuilt single executable file.

Using command `dotnet publish`, we can create a package of Framework dependent single file executable. This executable comprises of all dependencies required to run and its self-extracting:

```
dotnet publish -r win10-x64 -p:PublishSingleFile=true
```

We can also set property `PublishSingleFile` as `true` in `propertyGroup`, as shown below:

```
<PropertyGroup>
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

To learn more about single file executable, refer to Microsoft document on a single file bundle design: <https://github.com/dotnet/designs/blob/master/accepted/single-file/design.md>.

Assembly linking

.NET Core 3.1 uses the IL linker tool, which reduces the size of the application by scanning the unused libraries. It verifies the code and its dependencies and removes unused assemblies, which reduce the size of the app. For example, self-contained applications store code as well as all dependencies; it doesn't care whether the .NET framework installed on the host machine or not. Still, all .NET assemblies generally do not require to run application code. This tool trims those unused assemblies. We can set this as property in `PropertyGroup` of the project file:

```
<PropertyGroup>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

Note: The IL linker tool cannot be used in all the scenarios. For example, in the case of Dynamic loading or when we use reflection, in these cases, we will get exceptions because the IL linker tool can't find assemblies that are loaded dynamically, and those get trimmed out beforehand itself.

Read more about IL linker from Microsoft document: <https://github.com/mono/linker/blob/master/src/ILLink.Tasks/README.md>.

Tiered compilation

The tiered compilation is a default with .NET Core 3.1. .NET framework previously used to compile the code once. So, JIT compilation can provide anyone either a steady performance or start fast (reduced startup time). Both have its trade-offs, for example, if we want to reduce the application startup time, in this case, we need JIT compilation to be fast. We do not concentrate on code quality optimization, and if we are looking for steady performance, JIT will take time on startup, and it will create optimized code.

What if we get both? To achieve both steady-state as well as less startup time, we need two different ways of compilation. We can achieve this through Tiered compilation, which was introduced with .NET Core 2.1. Tiered compilation allows multiple compilations of the same code/method that can be swapped at runtime. With tiered compilation, we can select different techniques based on its purpose. As we discussed here, startup time reduction and steady-state performance; for both, we can have different techniques and use it as required. Hence, it is the benefit of tiered compilation; we can achieve both startup time reduction by doing a quick

compilation without code optimization, and later on, if the method is getting used multiple times, then optimized code gets generated on the background thread. A pre-compiled version of code is replaced by an optimized code for a steady-state.

Tiered compilation testing demo is shared by Microsoft on GitHub at location: https://github.com/aspnet/JitBench/blob/tiered_compilation_demo/README.md.

ReadyToRun images

ReadyToRun is known as R2R format. It is a form of **AOT (Ahead of Time)** compilation. R2R is useful for improving the startup time of the .NET Core application. We discussed above tiered compilation, and even before JIT compilation, ReadyToRun image reduces the efforts of JIT and makes a startup faster. ReadyToRun image size is significant because it contains IL code as well as native code.

Point to keep in mind is that ReadyToRun image format is available only when we publish a self-contained application that focuses on a specific runtime environment.

To publish a self-contained application for a specific runtime environment, use the following command:

```
dotnet publish -c Release -r win-x64 --self-contained
```

To set ReadyToRun format for the self-contained application, open project file and add property PublishReadyToRun, under PropertyGroup and set it true as shown in the following code:

```
<PropertyGroup>
  <PublishReadyToRun>true</ PublishReadyToRun >
</PropertyGroup>
```

Cross-platform/architecture restrictions

R2R (ReadyToRun) compiler doesn't support cross targeting. Publish command should run on the same environment for which the R2R image is created. Few exemptions for cross targeting are:

- Windows x86 can be used for the compilation of Windows ARM32 images
- Linux x64 can be used for the compilation of Linux ARM64 / ARM32 images
- Windows x64 can be used for the compilation of Windows ARM32 / ARM64 / x86 images

Runtime/SDK

Under this section, we have the following enhancements.

Build copies dependencies

Previously NuGet dependencies and other dependencies used to get copied only at the time of publishing by using the command: `dotnet publish`, but now all NuGet dependencies can be copied from NuGet cache to output build folder using build command: `dotnet build`.

Local tools

In previous releases of .NET Core, the installation of global tools was allowed. .NET Core has backed global tools since the very first release, but what if we need a tool on Local, within the context of a specific project or within certain directories on our computer.

To fulfill the need for local installation instead of global .NET, Core 3.1 came up with local tools. Using .NET Core 3.1, we can now install local tools as well; those are scoped to a specific directory. Local tools are introduced with .NET Core 3.1. The local tool is a special NuGet package that contains console applications and installed in our machine at a default location and coupled with a specific location on disk.

In our current directory, manifest file `dotnet-tools.json` is available, and local tools depend on it. This manifest file describes all available tools at the folder location or inside subfolders/ child folders. Local tools would be available to subdirectories also if it is installed at the directory level.

If we are sharing code, we should share the manifest file also so that the same tools can be restored and utilized by the code at the distributed location.

In case of a new project, we can create a .NET local tool manifest file by using the below command:

`dotnet new tool-manifest`

To install the tool locally, we can use the following command:

`dotnet tool install <tool_name>`

The above command is like what we use for installing global tools, just that we don't write `-g` at the end of the command.

To run the local tools, the command is like global tools; we just need to add prefix `dotnet` in command.

Local tools are an excellent way to make project-specific tooling. To make it available in the context of a project without a need to install it globally on our machine.

Note: Still, many local/global tools at NuGet.org is still referring to .NET Core 2.1 Runtime. For those tools, we need to install .NET Core 2.1 Runtime.

Smaller Garbage Collection heap sizes

.NET Core 3.1 is using less memory because the default heap size of the **garbage collector (GC)** is reduced. So, now .NET Core 3.1 works in a better way with containers because of less memory allocation. Previous versions used to allocate large heap per CPU and garbage collection used to happen based on memory usage versus available memory. It can cause out of memory. In .NET Core 3.1 memory is considered while creating heap.

Garbage Collection Large Page supports

Garbage collection now comes with setting GCLargePages. Using this setting, we can decide to give large pages on Windows. A significant page is a feature where the **OS (operating system)** can allocate memory greater than its native size, which is usually 4k. A significant page feature is a feature to increase the performance of applications that are requesting for large pages.

Opt-in feature

.NET Core 3.1 added a new feature called an opt-in feature. This feature allows our application to roll forward to the latest major version of .NET Core. Roll forward can also be controlled by using different configurations like:

- **Minor:** Minor is the default setting in case nothing is provided explicitly. LatestPatch policy used with requested minor version, and if requested minor version is not present, then roll forward to the lowest higher minor version.
- **LatestPatch:** Roll forward to the highest available patch version and disable roll forward of the minor version.
- **LatestMinor:** Roll forward to the highest/latest minor version even if the requested minor version is available.
- **LatestMajor:** Roll forward to highest/latest minor and significant version even if requested significant version is available.
- **Major:** Use Minor policy if requested Major version is available. If the version is missing, in that case, roll forward to the lowest higher major and its lowest minor version.
- **Disable:** This setting disables the roll forward feature, and the version will not upgrade to the latest. This setting is only suggested for testing to set the version fixed and will not upgrade to the latest.
- Feature bands upgrades will be "in place." So, if we have .NET Core 3.0.101 installed and we are now installing a new version .NET Core 3.0.102, in this

case, version 101 will be replaced by version 102. It is because both belong to the same feature band.

"in place" versioning is not going to replace "side by side" versioning. In continuation to the above example, now if we install .NET Core 3.0.202, it will not replace .NET Core 3.0.102, because these are two different feature bands.

IEEE Floating-point improvements

IEEE 754-2008 got published in August 2018, and it has significant revisions on IEEE 754-1985 floating-point standards. Few of them are 16 bit / 128-bit binary type, new operations, three decimal types, and recommended methods are added in the standards. To know more about IEEE revisions, read from <https://ieeexplore.ieee.org/document/4610935>.

.NET Core APIs are added/updated to obey the IEEE 754-2008 revision. The primary purpose of floating-point improvements is to support all necessary/essential operations, and .Net Core APIs are compliant with IEEE standards. Few main improvements and additions are:

1. Parse correctly and round the input of any length.
2. Parse correctly and do case-insensitive check.
3. Newly added math APIs:
 - a. `Math.BitIncrement(<floating-point>), Math.BitDecrement(<floating-point>)`
 - b. `CopySign`: Using this, we can return the value of one variable but with the sign of another variable. It corresponds to IEEE operations.
 - c. `FusedMultiplyAdd`: It performs multiply and adds as a single operation like $a + (b * c)$.

There are many improvements done to align with new IEEE standards. Few of them are listed above.

Built-in JSON support

JSON.NET is an open-source library for JSON serialization and deserialization. It is very well supported with .NET framework and ASP.NET, but the problem raised is that ASP.NET Core depends on JSON.NET and app, which we create is also coupled with JSON.NET, so it ties up our application to a specific version of JSON.NET. Now, if we want to upgrade to the latest version of JSON.NET or want to use any library which depends on some other version of JSON.NET than the one which framework version is using/supporting, this we cannot achieve quickly with ASP.NET Core.

.NET Core 3.1 introduces `System.Text.Json` namespace. This namespace contains classes that work with JSON data. These classes are built for:

1. **High performance:** Work with raw UTF8 format
2. **Less memory usage:** Minimized allocation using span data type to work with json data
3. High throughput

With improvements, there are few limitations also. As JSON.NET is evolved with years and covered a lot, many scenarios, like serializing enum values as strings instead of numbers is not supported by new classes under `System.Text.Json` namespace. These missing features will be included in future versions. New .NET Core version uses `System.Text.Json` by default, but still, we can install the JSON.NET NuGet package and keep using it, but if JSON requires simple operations, which is already available in `System.Text.Json` then go ahead with it. With `System.Text.Json`, version upgrade will be accessible, and new libraries are faster.

Json Reader

Let's take an example of the UTF8 JSON reader class for JSON parsing. We will read a JSON using the new `Utf8JsonReader` class. Let's create a JSON file first. We have created a JSON file which contains the detail about a book and its language, author, and many more:

```
{  
    "book": ".NET Core 3.1 What's new",  
    "language": "C#",  
    "authorDetail": {  
        "firstname": "Neha",  
        "lastname": "Shrivastava"  
    },  
    "isAvailableInMarket": true,  
    "tags": [ ".NET Core", "C#8", "New" ]  
}
```

Set the following settings for `example.json` file to copy the file in output directory on the build. Set **Copy to Output Directory** as **Copy if newer**:

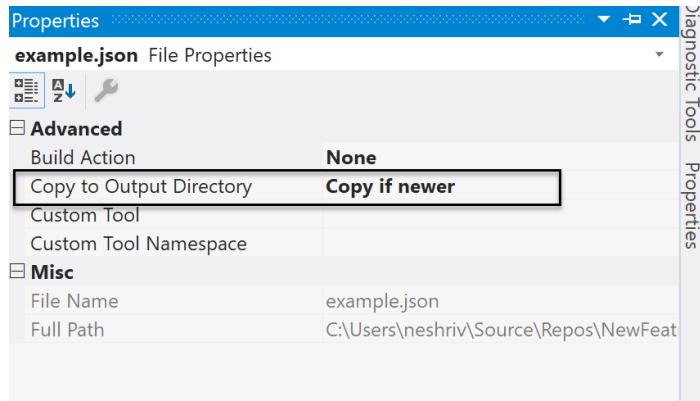


Figure 3.5: JSON file settings

`Utf8JsonReader` takes input as a read-only span of UTF-8 encoded text; it doesn't take files/streams directly as an input parameter. To create a span from the JSON file, we are first reading all bytes from `example.json` file using `File` class. It will result in an array of bytes:

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Text.Json;

namespace NewFeaturesOfCSharp8
{
    public static class BuiltinJsonSupport
    {
        public static void ReadExampleJson()
        {
            Console.WriteLine("using utf8 json reader class");
            var exampleJsonBytes = File.ReadAllBytes("example.json"); → Read all bytes of example.json file
            var exampleJsonSpan = exampleJsonBytes.AsSpan(); → Convert array of bytes into span
            var jsonexample = new Utf8JsonReader(exampleJsonSpan); → Utf8JsonReader need span input
            while (jsonexample.Read())
            {
                Console.WriteLine(GetValueType(jsonexample)); → Read Json and evaluate text
            }
        }
    }
}

```

Figure 3.6: `utf8JsonReader` example

We converted the bytes into a span by calling the `AsSpan` extension method, and then we passed `jsonSpan` as the input parameter to `Utf8JsonReader`. After the reader is initialized, we looped through our JSON data using a `while` loop. Whenever we call `Read()`, the reader will go forward to the next token in the JSON file data.

There are many types of `JsonTokenType` is defined. We can select output based on the token type. In this example, we are going to add a method that tells us the information about each token by its type. For example:

```
private static string GetValueType(Utf8JsonReader json) =>
    json.TokenType switch
    {
        JsonTokenType.StartArray => "Start Array",
        JsonTokenType.EndArray => "End Array",
        JsonTokenType.StartObject => "Start Object",
        JsonTokenType.EndObject => "End Object",
        JsonTokenType.PropertyName => $"Property : {json.GetString()}",
        JsonTokenType.Comment => $"Comment :{json.GetString()}",
        JsonTokenType.String => $"String :{json.GetString()}",
        JsonTokenType.Number => $"Number :{json.GetInt32()}",
        JsonTokenType.True => $"Boolean :{json.GetBoolean()}",
        JsonTokenType.False => $"Boolean :{json.GetBoolean()}",
        JsonTokenType.Null => $"Null",
        _ => $"No token :{json.TokenType}",
    };
}
```

We used a new C#8 switch expression for the token type. JSON reader instance reveals information about the current token. `TokenType` is the current token, and helper methods get the value of that token. In Switch expression, we added cases for a few token types which we may encounter. There are tokens for start/end object, start/end array, and for the property, and so on.

Switch expression is going to return a description of each type of token. We have called this method inside while loop. It will load the `example.json` file and loop through the tokens in this document and write information about each token in the console. Let's see how it works.

I have arranged JSON file and console output side by side for line up tokens with its value. We can see different tokens relate to what we would expect based on `example.json` file.

Start object then property name is "`book`", and the string value is ".NET Core 3.0 What's new". Similarly, "`language`", "`authorDetail`" then a new start object which

contains two properties "**firstname**" and "**lastname**" and its string value and so on and likewise starts array and end array:

```

using utf8 json reader class
Start Object
Property : book
String :.NET Core 3.0 Whats new
Property : language
String :C#
Property : authorDetail
Start Object
Property : firstname
String :Neha
Property : lastname
String :Shrivastava
End Object
Property : isAvailableInMarket
Boolean :True
Property : tags
Start Array
String :.NET Core
String :C#8
String :New
End Array
End Object

```

Schema: <No Schema Selected>

```

{
  "book": ".NET Core 3.0 Whats new",
  "language": "C#",
  "authorDetail": {
    "firstname": "Neha",
    "lastname": "Shrivastava"
  },
  "isAvailableInMarket": true,
  "tags": [".NET Core", "C#8", "New"]
}

```

Figure 3.7: JSON reader output

Json Writer

We have seen an example of `utf8JsonReader`, now let's have a look at JSON Writer. We will see how to write data using `utf8JsonWriter`.

`utf8JsonWriter` requires a buffer or stream for writing. The following screenshot displays both `Utf8JsonWriter` methods and its input parameter defined under `Utf8JsonWriter` class and `System.Text.Json` namespace:

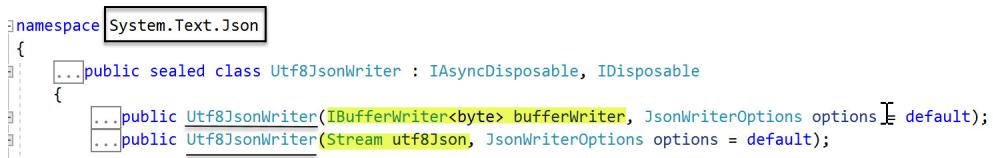


Figure 3.8: `Utf8JsonWriter` definitions

To set the input, we created an instance of `ArrayBufferWriter <byte>`, data can be written to this. We can also set the JSON alignment, which is an optional input parameter. If nothing is passed, it uses the default JSON Writer option. In our example, we are setting `JsonWriterOptions` to `Indented = true` so, the output will be indented to make it easier to read, each token in the next line, not in a single line.

After setting buffer and `JsonWriterOptions`, we passed both the parameters to the `Utf8JsonWriter` constructor. We can now create JSON data.

We have created a new method `WriteInJson` to populate writer which we have created:

```
public static class BuiltinJsonWriteSupport
{
    public static void WriteExample()
    {
        var align = new JsonWriterOptions
        {
            Indented = true
        };
        var buffer = new ArrayBufferWriter<byte>();
        using var examplejsonWriter = new Utf8JsonWriter(buffer, align);

        WriteInJson(examplejsonWriter);
        examplejsonWriter.Flush();
        var result = buffer.WrittenSpan.ToArray();
        var displayResult = Encoding.UTF8.GetString(result);
        Console.WriteLine(displayResult);
    }
}
```

As we were reading the token from JSON by the reader method, we can pass here tokens' value. In the `WriteInJson` method, we are building a JSON object, so, first, we created a start object token; this will add { for JSON and similarly end of object token, which will add }. So now we have a valid JSON file with open and close bracket:

```
private static void WriteInJson(Utf8JsonWriter examplejsonWriter)
{
    examplejsonWriter.WriteStartObject();
    examplejsonWriter.WritePropertyName("bookname");
    examplejsonWriter.WriteStringValue("Making your own json");
    examplejsonWriter.WriteStartObject("author");
    examplejsonWriter.WriteString("first", "Rishabh");
    examplejsonWriter.WriteString("last", "Verma");
    examplejsonWriter.WriteEndObject();
    examplejsonWriter.WriteEndObject();
}
```

We can now write a property name token inside this a value for that property. We could write a property name in its value in one place using `WriteString`. Also, we can write nested objects. We added an author property with an object value.

We have added some data, now let's present it in console. First, we need to tell our writer to flush its contents to the buffer, now we can take the output of our writer by using its buffer:

```
examplejsonWriter.Flush();
var result = buffer.WrittenSpan.ToArray();
var displayResult = Encoding.UTF8.GetString(result);
Console.WriteLine(displayResult);
```

It provides an array of bytes. What's convert that to a string, which we can now display in the console. We can also write this to a file. In below image we can see the console which displays indented JSON output and for better understanding correlated each token with its JSON:

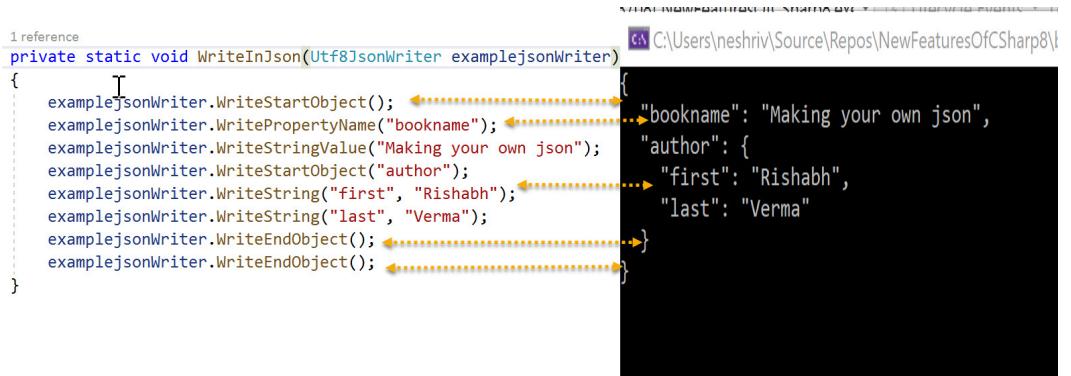


Figure 3.9: JsonWriter example

Json Serializer

We have read the JSON data using `utf8JsonReader` and wrote JSON data using `utf8JsonWriter`. Let's see now JSON Serializer/Deserializer method and how to use it:



Figure 3.10: Deserializer methods with overloads

The `Deserialize` method requires one of three things as an input, either a `utf8JsonReader` or a span of bytes containing JSON data or a string containing JSON Data.

In our example, we will read the example JSON file into a stream. We will pass our JSON text to the `Deserialize` method. Another optional parameter is `JsonSerializerOptions`; here, we will specify the `JsonNamingPolicy` and settled it to `CamelCase`. We did this because our property names are following `CamelCase`, and the default option is `Pascal`. We can pass additional options also to make it easier to read:

```
public static void RunDeserializer()
{
    var exampleJsonBytes = File.ReadAllBytes("example.json");
    var namingPolicy = new JsonSerializerOptions
    {
        PropertyNamingPolicy = JsonNamingPolicy.CamelCase
    };
    var book = JsonSerializer.Deserialize<Book>(exampleJsonBytes, namingPolicy);
    Console.WriteLine($"Book name:{book.BookName}");
}
```

Now, if we run, we will get the anticipated output. The JSON Serializer works in both ways. We can convert an object into JSON. JSON.NET is more robust, but the new built-in JSON Serializer/Deserializer will evolve with time as its new but works well with many scenarios.

HTTP/2 support

Many new APIs need new HTTP/2 protocol. Now ASP.NET also supports HTTP/2 protocol, and many services will start supporting this in the future. Keeping this in mind in .NET Core 3.1, support for HTTP/2 in `HttpClient` is added. To know more about HTTP/2, read it from <https://en.wikipedia.org/wiki/HTTP/2>.

Cryptographic Key Import and Export

AES-GCM and AES-CCM support are now added into .NET Core 3.0. These are implemented using `System.Security.Cryptography.AesGcm` and `System.Security.Cryptography.AesCcm` respectively. In .NET Core, **AE (Authenticated Encryption)** is the first algo that is added.

Now .NET Core 3.0 supports the import/export of private keys/asymmetric public keys from standard formats without the need for X.509 cert.

Summary

We have learned about new features and enhancements done as part of .NET Core 3.1, and we also created a WPF application using .NET Core 3.1 application. Used windows template provided by Visual Studio. We have seen build-in JSON support and its methods reader, writer, and Serializer/Deserializer. We also discussed deployment types and default deployment type for .NET Core 3.1. We have learned about the benefits of local tools and commands to install it.

There are many more small features added in .NET Core 3.1 like Japanese calendar support, Linux related features, which you can learn through the official Microsoft documentation site and blogs. One useful link is shared in the below exercise.

Exercise

1. What is self-contained deployment?
2. Which command is used for making self-extracting single file executable? What is the other way of setting it?
3. What IL Linker tool does?
4. What is a tiered compilation, and how it works?
5. How are local tools different from global tools?
6. What are the pros and cons of built-in JSON and JSON.NET?

Read this blog to understand .NET Core 3.1 <https://devblogs.microsoft.com/dotnet/announcing-net-core-3-1/>.

CHAPTER 4

Demystifying Threading

"Make a system that even a moron can use, and a moron will use it. Don't underestimate people's ability to break your code."

Threading is a fascinating and exciting topic, both for discussions as well as for usage and implementation. If you ever work on any enterprise-grade software application, the chances are that you would need to leverage threading. In this chapter, we will discuss threads and tasks in length and build a solid foundation on threading so that we can take off in the world of threading with confidence.

Structure

- Why threading?
- What is threading?
- Thread (exception handling and limitations)
- ThreadPool (exception handling and limitations)
- ThreadPool in action
- Tasks
- TaskCreationOptions

- Exception handling with Tasks
- Cancellation
- Continuations
- WhenAll, WhenAny
- Task Scheduler
- Task Factory
- Summary
- Exercise

Objectives

By the end of this chapter, the reader should be able to understand:

- Threading and need for it
- Threads and their limitations
- ThreadPool and why it should be used
- Task, Task Cancellation, Task Continuations
- Task Scheduler and Task Factory

Why threading?

Before we dive in, it's imperative to understand the need for threading. While you read this book, your body is doing multiple tasks simultaneously like digesting the food that you ate earlier, pumping the blood to various organs of your body, inhaling oxygen, exhaling carbon dioxide, and so on. All these functions are needed for your body to function correctly and are happening at the same time. The reason that all these activities can happen at the same time is that these functions are being anchored by different subsystems and organs in the body like digestive system breaks down the food and extract nutrients; while the heart pumps the blood and respiratory system takes care of inhaling and exhaling and so on. Likewise, most of the enterprise applications and software that you would develop or code would have a requirement of conducting multiple tasks simultaneously. To achieve this in the world of Windows and .NET, threading is the way to go, which makes threading a must-know skill.

To demonstrate threading in action, let's consider an email client application. Being on Windows 10, that program happens to be Outlook in my machine. While I draft a new email to my team, Outlook may be checking for new email(s) that may be sent to me, or maybe archiving old email at the same time. This can be achieved by

threading. So threading is essential! Let us see a few of the most common scenarios where threading finds a great use case:

1. **Developing a responsive user interface (UI):** GUI based Windows desktop applications built on Windows Form, **Windows Presentation Foundation (WPF)** or **Universal Windows Platform (UWP)** or Xamarin, and so on, have to deal with high CPU consuming operations or operations that may take too long to complete. While the user waits for the operation to complete, the application UI should remain responsive to the user actions. You may have seen that dreadful "Not responding" status on one of the applications. This is a classic case of the Main UI thread getting blocked. Proper use of threading can offload the main UI thread and keep the application UI responsive.
2. **Handling concurrent requests in server:** When we develop a web application or Web API hosted on one or many servers, they may receive a large number of requests from different client applications concurrently. These applications are supposed to cater to these requests and respond in a timely fashion. If we use the ASP.NET / ASP.NET Core framework, this requirement is handled automatically, but threading is how the underlying framework achieves it.
3. **Leverage the full power of the multi-core hardware:** With the modern machines powered with multi-core CPUs, effective threading provides a means to leverage the powerful hardware capability optimally.
4. **Improving performance by proactive computing:** Many times, the algorithm or program that we write requires a lot of calculated values. In all such cases, it's best to calculate these values before they are needed, by computing them *in parallel*. One of the great examples of this scenario is 3D animation for a gaming application.

Now that we know the reason to use threading, let us see what it is.

What is threading?

Let's go back to our "human body" example. Each subsystem works independently of another, so even if there is a fault in one, another can continue to work (at least to start with). Just like our body, the Microsoft Windows operating system is very complex. It has several applications and services running independently of each other in the form of processes. A process is just an instance of the application running in the system, with dedicated access to address space, which ensures that data and memory of one process don't interfere with the other. This isolated process ecosystem makes the overall system robust and reliable for the simple reason that one faulting or crashing process cannot impact another. The same behavior is desired in any application that we develop as well. It is achieved by using threads, which are the basic building blocks for threading in the world of Windows and .NET.

If I have a look at the Windows Task Manager (*Ctrl + Shift + Esc*) and go to the **Performance** tab, below is how it looks like:

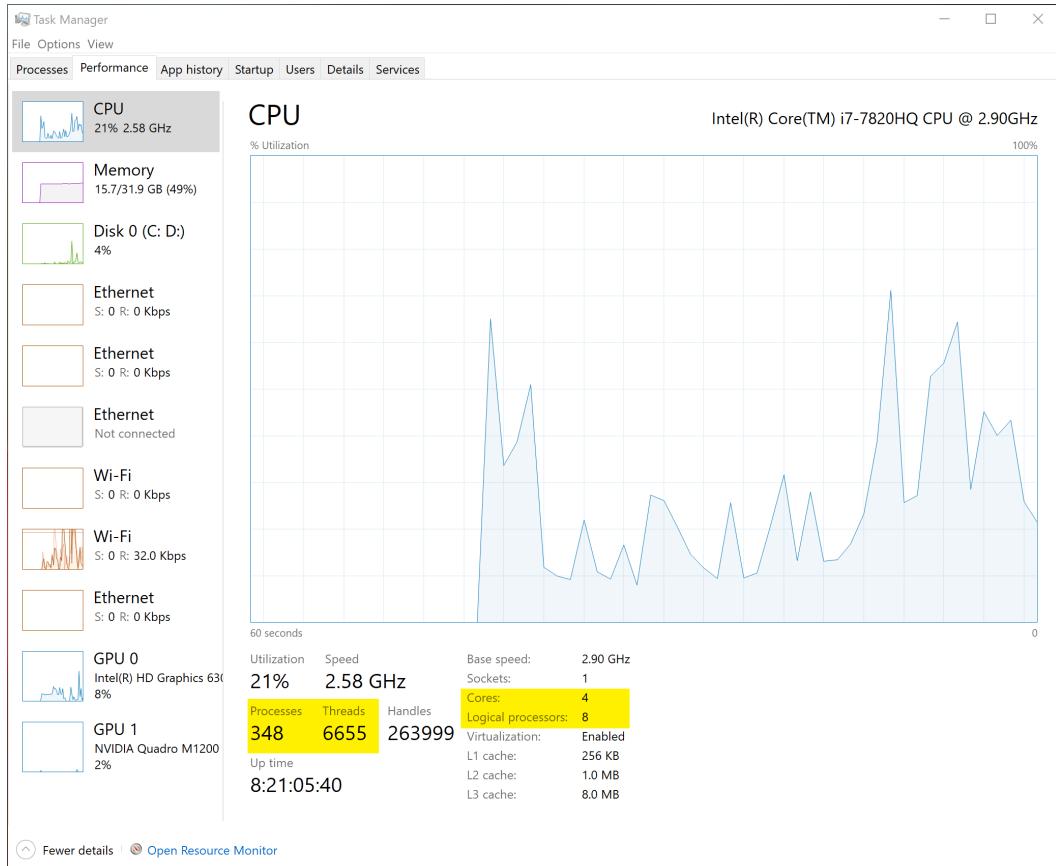


Figure 4.1: Task Manager

As highlighted in the image, there are 4 Cores and 8 Logical processors in my machine. You may see different values in your machine, depending upon your machine configuration.

Essentially, it tells us that my **central processing unit (CPU)** has four cores, and its hyper-threaded (we will discuss it shortly below), and so makes the operating system think that there are eight processors, so we see logical processors as 8. Think of it as cores represent the hardware side of things and are actual processors physically present in the chip. In contrast, logical processors represent the software side of things and equal the number of processors that the operating system thinks there are on your chip.

At any given instance of time, the number of processors determines how much work your machine can do at the same time, so the race is on to increase the number

of processors. The following terms are frequently encountered which discussing CPU:

- **Hyper-Threaded CPU:** Also known as HT CPU, technology was invented by Intel Corporation for parallel computing. This technology allows a single processor to appear as multiple logical processors from the perspective of software. HT-CPU enables the operating system and applications to schedule multiple threads on a single physical processor at a time. Eight logical processors shown above are the result of hyper-threading.
- **Multi-Core CPU:** Earlier, computers had one CPU, which had one core, that is, one processing unit. To boost the performance, the CPU manufacturers started adding more cores to the CPU, so came CPU with two core, four-core, eight-core, and so on called the **dual-core**, **quad-core**, **octa-core** respectively. But unlike hyper-threading technology discussed above, there are that many physical processors available on the hardware chip. The four cores in the preceding screenshot show that my machine is quad-core; that is, it has four cores.
- **Multiple CPU or Multiprocessor:** How about having multiple CPU chips in the motherboard? That is precisely what multiple CPU technology is, which was tried before hyper-threading or multi-core technology came into existence. Multiple CPU technology requires the motherboard to be modified to accommodate and use multiple CPU chips. Multiple CPU needs more power, cost, and cooling as well, so it is not very common. Generally, only high-end servers, or gaming machines or supercomputers use multiple CPU technology.

Now that we have discussed CPU technologies let's go back to our image, which also tells us that, I have 348 processes, which have 6655 threads running and utilizing 21% of the total CPU and 49% of the total RAM. Wow! It means an average of 19 threads per process. Let us discuss thread.

Thread

Thread is defined as a light-weight process. So, it is a component of the process. Thread is the basic unit that is allocated processor time by the operating system. Thread is a Windows devised concept primarily to virtualize the CPU and provide an execution path independent of others.

Multiple threads can exist in a process. When a process is created, it is allocated virtual address space. Threads need to communicate with other threads and share data with other threads, so each one has access to a shared heap. Therefore, each thread in the process shares this address space.

Threads can execute independently, so each one has its stack. Each thread has a scheduling priority. The operating system is tasked to ensure all threads are allocated

processor time. It may well be the case that one thread is executing a long-running operation, and the operating system needs to allocate the processor time slice to some other thread in a scheduled fashion, based on thread priorities, so one thread needs to pause. In contrast, other thread does work and so on. When the turn of the same thread comes again, the operating system will again allocate the processor time slice to the thread, and it should resume its operation from where it last paused. To enable this, resumption of operation from the paused state, each thread maintains a set of data structures to persist the context at the time of pause. This context has all the necessary information required to resume execution of operation along with CPU registers and stack information.

We see that for running a thread, CPU time slice is needed, so at any given instance of time, the number of concurrently running threads would be equal to the number of processors in the machine.

You can check the number of processors programmatically by getting the `System.Environment.ProcessorCount` property. When the number of threads is more than several processors, the operating system will schedule the CPU time slice to threads, and so there would be a context switch between threads, and hence performance may take a hit. Since in context switch, the thread needs to maintain its state in data structures so that they can resume the operation from the same place, when they get the CPU time slice again, the data structures would be allocated in RAM. The key takeaway from this discussion is that thread creation is expensive and, like buying any expensive stuff, should be done only after thorough deliberation. The next image depicts a high-level representation of thread:

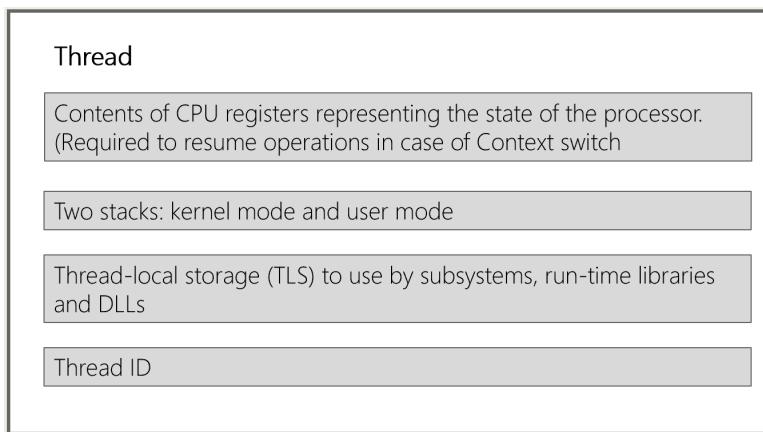


Figure 4.2: Representation of a Thread

By default, any .NET /.NET Core program starts with a single thread, which is called the Main thread or Primary thread. As the program runs, it can spawn multiple threads to execute code concurrently as needed. All these new threads are called Worker threads.

In .NET Core, `System.Threading.Thread.dll` assembly contains the APIs to leverage the might of threading (apart from few other assemblies like `mscorlib.dll`, `netstandard.dll`) and `System.Threading` is the namespace that we would need to import to use it. Let's get started with code to see threads in action. To do this, we will create a simple console app (.NET Core) in Visual Studio 2019. For the continuity of reading, the code snippet from the `Program.cs` is pasted below:

```
class Program
{
    static void Main(string[] args)
    {
        WriteToConsole();
        Console.ReadLine();
    }

    static void WriteToConsole()
    {
        string name = string.IsNullOrWhiteSpace(Thread.
CurrentThread.Name) ? "Main Thread" : Thread.CurrentThread.Name;

        if (string.IsNullOrWhiteSpace(Thread.CurrentThread.Name))
        {
            Thread.CurrentThread.Name = name;
        }

        Console.WriteLine($"Hello Threading World! from {Thread.
CurrentThread.Name}");
        Console.WriteLine($"Managed Thread Id: {Thread.
CurrentThread.ManagedThreadId}");
        Console.WriteLine($"IsAlive: {Thread.CurrentThread.
IsAlive}");
        Console.WriteLine($"Priority: {Thread.CurrentThread.
Priority}");
        Console.WriteLine($"IsBackground :{Thread.CurrentThread.
IsBackground}");
        Console.WriteLine($"Name: Thread.CurrentThread.Name");
        Console.WriteLine($"Apartment State:{Thread.
```

```
CurrentThread.ApartmentState.ToString()});  
        Console.WriteLine($"IsThreadPoolThread :{Thread.  
CurrentThread.IsThreadPoolThread}");  
        Console.WriteLine($"ThreadState :{Thread.CurrentThread.  
ThreadState}");  
        Console.WriteLine($"Current Culture : {Thread.  
CurrentThread.CurrentCulture}");  
        Console.WriteLine($"Current UI Culture : {Thread.  
CurrentThread.CurrentCulture}");  
        Thread.Sleep(5000);  
    }  
}
```

We write a simple static method named `WriteToConsole`, where we just write the properties of executing thread to the console window. To get the thread executing the method, we use the `Thread` class' `static CurrentThread` property. All the properties are intuitive and easily understandable, but we shall discuss various properties during this discussion. The code in the repository has comments to make the code more understandable.

When we run this code in Visual Studio 2019 by pressing the *F5* key (**Start Debugging**), we get the output printed in the console window. For reference, the output would be:

```
Hello Threading World! From Main Thread  
Managed Thread Id: 1  
IsAlive: True  
Priority: Normal  
IsBackground: False  
Name: Main Thread  
Apartment State: MTA  
IsThreadPoolThread: False  
ThreadState: Running  
Current Culture: en-US  
Current UI Culture: en-US
```

It tells us the properties of the Main Thread that runs the console app.

The name of `Main Thread` was `null` by default; we programmatically set the name of the thread as `Main Thread`. The `Name` property can be set only once and has

the default value of `null`. Attempting to set the `Name` property again after it is set once, would throw an `InvalidOperationException`. The `Name` property is useful for the developers to debug the multithreaded code as they can identify what thread executed the code.

`ManagedThreadId` is of type `integer` and is used to identify a thread within a process uniquely. It's a read-only property and cannot be set programmatically.

Since this thread is executing the program, it's no surprise that the `IsAlive` property is `true`, but had the thread been terminated or aborted, this property would have been `false`.

The default scheduling priority of a thread is `Normal`. Though it can be set to other priorities based on the enum `ThreadPriority`, like `AboveNormal`, `Highest`, `BelowNormal`, and `Lowest`, the operating system may or may not choose to honor the priority of the thread.

`IsBackground` property of the thread indicates if the thread is a foreground thread or a background thread. A thread can either be a background thread or a foreground thread. A foreground thread prevents the process from terminating unless they are terminated. Once all foreground threads terminate, background threads, if any running in the process is stopped and may not run to completion. No exception is thrown when the background thread is ended this way.

By default, a thread created by `Thread` class constructor is a foreground thread, that is, it has `IsBackground` value set as `false`. The Main Thread under discussion is also a foreground thread. Hence, its `IsBackground` property is `false`. A programmer is free to change the background thread to the foreground or vice versa programmatically by changing the `IsBackground` property value. We will discuss `ThreadPool` a little later in the chapter, but while we are at it, it's handy to know that all thread pool threads are background threads, and so have their `IsBackground` property set to `true`.

`ApartmentState` property is now obsolete and is just shown for the sake of completeness. It's a read-write property and is used to get or set the apartment state of a thread. As per Microsoft documentation, Apartment is a logical container in the process for objects sharing the same thread access requirements. All the objects in the same apartment can receive calls from any threads in the apartment. It can have one of the three possible values in `ApartmentState` enum, that is, **MTA (Multithreaded apartment)**, **STA (Single-threaded apartment)**, or `Unknown`. .NET and .NET Core doesn't use apartments, and CLR requires the developer to ensure thread safety while accessing shared resources. The apartment is a COM concept and has been kept in .NET (Core) for COM interop cases.

`IsThreadPoolThread` property indicates if the thread belongs to the CLR `ThreadPool` or not. For the `Main` thread, this value would be `false`.

`ThreadState` is a read-only property of `thread` that indicates the state of the thread. It can have one of the values of `ThreadState` enum.

`CurrentCulture` and `CurrentUICulture` of the thread are read-write properties that are used to get or set the current culture and current UI culture of the thread, respectively. For the main thread of execution, it displays my machine's culture, that is, en-US. However, based on the culture set in your machine, you may see a different culture value displayed on the console. It is important to note that in .NET Core, reading/writing these properties is not supported by another thread. Doing so would throw an `InvalidOperationException`, as we will see shortly. In such cases, it is advised to use `CultureInfo.CurrentCulture` property and `CultureInfo.CurrentUICulture` properties, respectively.

`Sleep` is a `static` method on `Thread` class, which makes the current thread of execution sleep or suspend for a specified number of milliseconds. In the above sample, the `Main` thread would sleep for 5 seconds.

At this point, if we insert a breakpoint on the line `Console.ReadLine();` of the `Main` method and run the program with debugging (`F5`), the method would execute and wait on this line. In the Visual Studio, click **Debug** → **Windows** → **Threads**. It would open the **Threads** window in the Visual Studio, which is of great help in debugging multithreaded applications. The **Threads** window shows that there is only one thread running in the process. It also displays the **Managed ID**, **Category**, **Name**, **Location**, **Priority** of the thread, as shown in the following screenshot:

The screenshot shows the 'Threads' window in Visual Studio. The window title is 'Threads'. At the top, there is a search bar labeled 'Search:' and a 'Group by:' dropdown set to 'Process ID'. Below the header, there is a table with columns: ID, Managed ID, Category, Name, Location, Priority, Process Name, and Process ID. A tooltip above the table says '^ Process ID: 10656 (1 thread)'. The table contains one row with the following data: ID: 10144, Managed ID: 1, Category: Main Thread, Name: Main Thread, Location: ThreadingBasics.dll!ThreadingBasics.Program.Main, Priority: Normal, Process Name: ThreadingBasics.exe (id = 10656), and Process ID: 10656.

	ID	Managed ID	Category	Name	Location	Priority	Process Name	Process ID
[^] Process ID: 10656 (1 thread)								
▼	10144	1	Main Thread	Main Thread	ThreadingBasics.dll!ThreadingBasics.Program.Main	Normal	ThreadingBasics.exe (id = 10656)	10656

Figure 4.3: Thread Window

Now, let us tweak our `Main` method to create a couple of threads and execute the same method `WriteToConsole` as shown in the following code:

```
static void Main(string[] args)
{
    // Main Thread
    WriteToConsole();
    // Thread 1
    Thread thread1 = new Thread(WriteToConsole) { Name =
    "Thread1" };
    thread1.Start();
    // Thread 2
```

```

        Thread thread2 = new Thread(WriteToConsole) { Name =
    "Thread2" };

        thread2.Start();
        Console.ReadLine();

    }

```

`Thread.Start` creates a new thread and runs the method passed to it as a parameter in its constructor. So, in the above code, `Main Thread` runs the method `WriteToConsole` and then creates two threads, `Thread1` and `Thread2`, which run the same method concurrently. If we insert the breakpoint again at the same line of code and see the **Threads** window, we will now see three threads, one main thread and two new threads, `Thread1` and `Thread2`.

Note that the category of the newly created threads is worker thread, so the newly spawned threads from `Main` thread are called **Worker threads**, and since we named them, we can see their name displayed in the **Threads** window as well. The location shows what code is being executed by that thread, that is, the stack trace. The next image (*Figure 4.4*) shows three threads in the **Threads** window:

ID	Managed ID	Category	Name	Location	Priority	Process Name	Process ID
Process ID: 7736 (3 threads)							
7632	1	Main Thread	Main Thread	✓ ThreadingBasics.dll!ThreadingBasics.Program.Main	Normal	ThreadingBasics.exe (id = 7736) 7736	
936	5	Worker Thread	Thread1	✓ System.Private.CoreLib.dll!System.Threading.ThreadHelper.ThreadStart	Normal	ThreadingBasics.exe (id = 7736) 7736	
10656	6	Worker Thread	Thread2	✓ System.Private.CoreLib.dll!System.IO.StreamWriter.WriteLine	Normal	ThreadingBasics.exe (id = 7736) 7736	

Figure 4.4: Three threads in Threads window

Note: Depending upon when the threads start running, you may or may not see three threads in the **Threads** window directly when the breakpoint is hit. If you don't see three threads, it may mean that either the new threads haven't started yet or have completed their work and terminated. To increase the chances of seeing, we have added the code `Thread.Sleep(5000)`; which keeps the threads in the blocked state for five seconds and hence you can continue the debugging (by either pressing `F5` or clicking **Play** button icon, that is, continue in the Visual Studio command bar) and immediately press (`Ctrl + Alt + Break`) to break the execution or by clicking pause icon in the Visual Studio command bar to break the execution. If your threads are running, you should now see the three threads in the **Threads** window, as shown in *Figure 4.4*.

Let us have a look at the output of this program in Console, shown below:

Hello Threading World! from Main Thread

Managed Thread Id : 1

IsAlive :True

Priority :Normal

```
IsBackground :False
Name :Main Thread
Apartment State :MTA
IsThreadPoolThread :False
ThreadState :Running
Current Culture : en-US
Current UI Culture : en-US
Hello Threading World! from Thread2
Hello Threading World! from Thread1
Managed Thread Id : 5
Managed Thread Id : 6
IsAlive :True
IsAlive :True
Priority :Normal
Priority :Normal
IsBackground :False
Name :Thread1
IsBackground :False
Name :Thread2
Apartment State :MTA
IsThreadPoolThread :False
Apartment State :MTA
IsThreadPoolThread :False
ThreadState :Running
ThreadState :Running
Current Culture : en-US
Current Culture : en-US
Current UI Culture : en-US
Current UI Culture : en-US
```

We find that apart from name and identifier, all the other property values of these two newly created threads are the same as that of Main Thread. However, we notice that the sequence of the values printed after the Main Thread completed is not predictable. It is because both Thread1 and Thread2 are running at the same time, and based on the scheduling by the operating system will get CPU time slice and execute the code to display the output, which is not predictable.

`System.Console` type on which `WriteLine` method is defined is designed thread-safe by the .NET team, so calling this method from multiple threads doesn't have unwarranted results.

Thread-safe code is the code that can be called from multiple threads without any unwanted or unintended interaction between the threads and provides predictable output. Therefore, we as developers should ensure that in any multithreaded program, any piece of code that can be called by multiple threads should be thread-safe and that multiple threads entering or executing shared code or shared resources among them don't result in unexpected output. To achieve this, we need to apply synchronization, which ensures that only one thread enters the critical region and executes the code. We will discuss thread synchronization in *Chapter 7*.

The great thing about .NET Core is that it is open source so that we can see the source code online. You can browse the URL <https://source.dot.net/> search for a type and see its source code. It helps us to see and understand the code better. If we search for `thread`, we can browse the source code of `Thread` class and see all its fields, constructors, properties, and methods. Following are a few of the worth discussing aspects of `Thread` class:

- It derives from the `abstract` class `CriticalFinalizerObject`, which ensures that all the finalization code in `Thread` class is marked as critical (hence critical finalizer object). So CLR guarantees that this code would be given full opportunity to execute even if the thread gets aborted or CLR unloads the application domain. An application domain is a logical container of assemblies and ensures isolation within a process.
- It has a `static` property `CurrentPrincipal` of type `IPrincipal` which returns the principal object that represents the security context of the user under which the code is running. It will return `null` if the principal object is not set.
- It has a `static` property `CurrentThread` that returns the currently executing thread. We used this property in the above sample to fetch the currently running thread.
- It has multiple constructors accepting `ThreadStart` and `ParameterizedThreadStart` types as a parameter, as shown in the following code block:

```
public Thread(ThreadStart start)
public Thread(ThreadStart start, int maxStackSize)
public Thread(ParameterizedThreadStart start)
public Thread(ParameterizedThreadStart start, int maxStackSize)
```

Both `ThreadStart` and `ParameterizedThreadStart` are delegates defined in `System.Threading` namespace as shown below:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object? obj);
```

Based on the method that needs to be invoked by the thread, corresponding constructor overload should be used. `ThreadStart` is a **delegate**, that is, a method pointer so it can be a **static** or instance method of the same or different classes.

- It has an instance method `Start`, to spawn a new thread.
- It has an instance method `Join`, which is a synchronization method that blocks the thread until the thread terminates. Some overloads accept time as a parameter.
- It has a **static** method named `Sleep`. This method can be used to suspend the thread for a specified amount of time. During this time, the thread is in the blocked state.
- It has an instance method `Abort`. Calling this method raises `ThreadAbortException` in the executing thread and eventually terminates the thread. It is the functioning in .NET. In .NET Core, calling this method would raise `PlatformNotSupportedException` as this API is not supported.
- It has a **static** method `Yield`, which, as the name suggests, yields the execution to another thread. The thread to which execution is yielded is chosen by the operating system. If this method returns true, it means the operating system switched the execution to another thread, else it returns `false`.

The last line of our `Main` method above is `Console.ReadLine()`; this line of code ensures that the console doesn't exit immediately after executing the code and waits for user input so we can see the output on the console. Since `Main` Thread is executing the `Main` method, so the `Main` Thread keeps on waiting on `Console.ReadLine()` after executing the `WriteToConsole` method and starting two threads. Like `Console.ReadLine()`, if any operation keeps waiting for something to happen like, for the user to enter input, or mouse to click, or for the data to download from network, or for the timer to get due; these operations are called **I/O (Input Output)** bound operation. On the other hand, if any operation keeps doing a CPU intensive work, then that operation is called CPU bound operation like performing a heavy calculation.

Exception handling

Continuing with the preceding code example, let's tweak our `WriteToConsole` method to throw an exception by adding `throw new InvalidOperationException();` as the last line of this method. It is to simulate the scenario in which exception occurs in the method code. We will also modify the `Main` method to insert `try...`

catch block to catch the exception. The complete sample can be seen in the `IncorrectExceptionHandling` project inside *Chapter 4* in the code repository of the book. Following is the relevant code after throwing an exception in the method:

```
static void Main(string[] args)
{
    try
    {
        // Thread 1
        Thread thread1 = new Thread(WriteToConsole) {
            Name = "Thread1" };
        thread1.Start();
        // Thread 2
        Thread thread2 = new Thread(WriteToConsole) {
            Name = "Thread2" };
        thread2.Start();
    }
    catch (Exception ex)
    {
        // Code never reaches here!
        Console.WriteLine($"An exception occurred while
creating threads {ex.Message}");
    }

    Console.ReadLine();
}
```

Since the thread is an independent execution path, exception handling code in `Main` `Thread` would not catch the exception thrown in any other thread. So, in the above code, when an exception is thrown in the method, it is not caught, and this unhandled exception causes the program to terminate. The correct way to handle the exception is to handle exception in the thread execution path itself, so exception handling code should be moved in the `WriteToConsole` method. This implementation can be seen in the `ExceptionHandling` project in *Chapter 4* folder of the repository.

Limitations

Though from the above discussion, threads might appear easy to create and provides a great way to execute the code concurrently. However, it comes with its own set of limitations and challenges. They are:

- Thread creation is expensive as it incurs space and time overhead. As we discussed earlier in our discussion, when context switch happens between threads (which is inevitable), the thread needs to pause its execution and save its context, which contains its properties as well as CPU registers in the thread kernel object data structure. This data structure needs to be created for every thread. Apart from this, a stack containing arguments and variables passed to the methods are allocated as well. This, by default, takes approximately 1MB for each thread. This memory overhead has nothing to do with the user program but is just needed for the threads to work.
- Using threads in too many places in an application makes the code less readable, complex to understand, debug, and may introduce performance issues.

Threads need to be used with caution and great understanding. My architect shared one of his experiences with threading from the times when he was a young developer. He was once working on a server-side enterprise application that had to do the batch processing of requests. It had several processing steps, so processing each request took considerable time. He wanted to write a very scalable, performant, fast, and highly concurrent system, so the young and enthusiastic developer inside him used threads to process the requests. He would create a new thread for each request, and then whatever further processing steps could be done in parallel, he would create new threads for them. All worked great in his local development machine and even in test and UAT environments with the expected load. Wow! Wonderful. The code got promoted to production and worked well for some time. One beautiful day, a customer came back stating that the system is down, and no requests could be processed.

As it turned out after investigation, that there was an unusually high number of requests during the week. They kept increasing over time, so a vast number of threads were created (remember that at any given point of time, the number of simultaneously running threads is equal to the number of cores in the machine) which took up all the resources and memory and the process started experiencing out of memory exceptions with the subsequent requests. Restarting the process fixed the issue for the time being. But this experience highlights that the thread life cycle needs to be effectively managed else; it may have unwanted ramifications.

To save the developers from these complexities, CLR has a `ThreadPool`, which does the life cycle management of threads in an optimized way. Let's discuss it.

ThreadPool

We discussed that creating and managing threads is an expensive operation with time and memory overhead and incur performance cost on account of context switching. .NET / .NET Core framework also needs to manage the threads for various framework

APIs, so CLR has a `ThreadPool`, which abstracts all these expensive and relatively complicated operations from the developers. The great news is that developers can use the `ThreadPool` class to leverage `ThreadPool` in their code. The pool has a queue where a request to perform an asynchronous operation can be enqueued. The `ThreadPool` would then dequeue the request and assign a `ThreadPool` thread to process that request. While a thread that we created earlier using `Thread` class executes an operation and terminates, following which the resources need to be reclaimed, `ThreadPool` threads complete the operation assigned to them and return to the `ThreadPool`, sit idle and wait for responding to a new operation. `ThreadPool` threads are not destroyed! It saves time and doesn't incur a performance hit.

When your application starts, it runs as a process. The process loads the CLR/CoreCLR, depending upon the executable. CLR is loaded for .NET executable while CoreCLR is loaded for .NET Core applications.

CLR/CoreCLR has a `ThreadPool`. To start with, `ThreadPool` doesn't have any threads. Based on several requests queued by the application to the `ThreadPool`, it would create the threads on demand. Operations that are queued to the `ThreadPool` would be assigned to a `ThreadPool` thread. The thread would complete the assigned task, return to the pool in a suspended state, and pick up other enqueued tasks if any. `ThreadPool` is self-adjusting, so if we enqueue a lot of requests for a substantial time, it may end up creating several threads to meet this need. Later, when the load reduces, and there are no more requests to be processed, it may kill the threads.

The `System.Threading.ThreadPool` class has the APIs that we can use to leverage `ThreadPool`. The important ones are:

```
ThreadPool.QueueUserWorkItem(WaitCallback)
ThreadPool.QueueUserWorkItem(WaitCallback, Object)
ThreadPool.QueueUserWorkItem<TState>(Action<TState>, TState, Boolean)
```

The `WaitCallback` parameter above is a `delegate` and is defined as:

```
public delegate void WaitCallback(object? state);
```

where the `state` is the object that contains the information for the `callback` method and can be `null` as well. Let's see this in action. The same sample can be seen in the `ThreadPool` project in *Chapter 4* folder of the code repository of the book:

```
static void Main(string[] args)
{
    System.Threading.ThreadPool.
    QueueUserWorkItem(WriteToConsole, "ThreadPoolThread");
    Console.ReadLine();
}
```

```
static void WriteToConsole(object state)
{
    string name = string.IsNullOrWhiteSpace(Thread.
CurrentThread.Name) ? state.ToString() : Thread.CurrentThread.Name;
    if (string.IsNullOrWhiteSpace(Thread.CurrentThread.Name))
    {
        Thread.CurrentThread.Name = name;
    }

    // Other code not shown for brevity.
}
```

The program executes the same `WriteToConsole` method, but this time via a `ThreadPool` thread. This example demonstrates how we can enqueue a method to be executed by `ThreadPool` thread. The output would look like the earlier program output. However, two properties in the output are worth seeing:

`IsBackground`

`IsThreadPoolThread`

Both these properties are set with value `True`, as the method is executed by `ThreadPool` thread, which is always a background thread.

As we see, the above three methods of the `ThreadPool` have the name `QueueUserWorkItem`, and these three are just overloads. The name "Queue user work item" makes it clear that `ThreadPool` makes use of `Queue`, where we can enqueue a task that needs to be processed by the `ThreadPool` thread. `ThreadPool` has an intelligent thread management algorithm that knows best as to when a new thread should be created, or an existing thread should be reused.

All the threads created in `ThreadPool` are background threads, that is, have their `IsBackground` property set to `true`. Hence it is an excellent time to recall that if the application has multiple tasks queued to `ThreadPool` but doesn't have an active foreground thread, all the background threads, as well as the application, would terminate as at least one active foreground thread is required to keep the application alive and running.

`ThreadPool` is designed and optimized for several small running operations. If we must perform a long-running operation, creating a manual thread is still a preferred and recommended way to go about it. There are two types of threads available in the `ThreadPool`, input/output (I/O) threads, and worker threads. As

their name states, they are used for asynchronous I/O processing and CPU intensive operations, respectively. A high-level representation of `ThreadPool` is shown in the next screenshot (*Figure 4.5*):

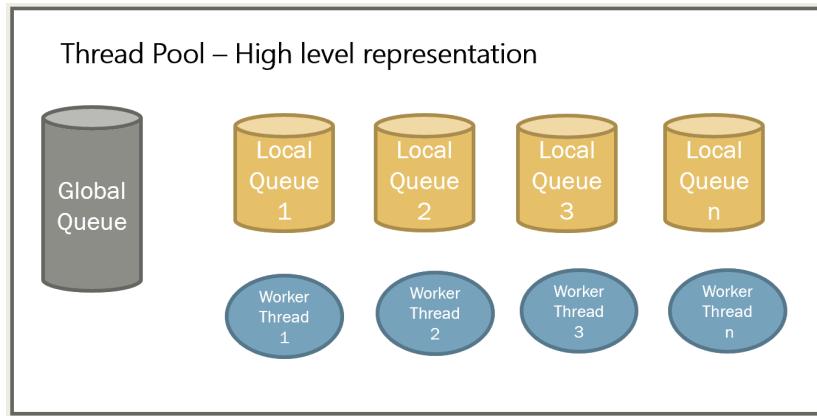


Figure 4.5: Visual representation of ThreadPool

We see that the `ThreadPool` has a global queue and number of worker threads (also non-worker threads not shown in *figure 4.5*). Each of the worker threads also has its local queue. When we queue a request to `ThreadPool` by using the `QueueUserWorkItem` method or when a `Timer` class executes a work item, it gets added to the global queue. The worker threads will then dequeue these work items from the global queue; that is, all the worker threads will try to dequeue the work item to process them in **first-in, first-out (FIFO)** fashion.

Since multiple worker threads are trying to dequeue from the same global queue, there is a synchronization lock applied to the queue, to ensure that a work item is processed by only one work item. Note that local queues of worker thread are not discussed yet. They come into the picture via `Task` infrastructure, which we will discuss later in the chapter, under the section `TaskScheduler`.

`ThreadPool` exposes the API to get and set the minimum and maximum threads in the `ThreadPool`.

To get the number of threads in `ThreadPool`, we have the following APIs:

```
public static void GetMaxThreads (out int workerThreads, out int completionPortThreads);  
  
public static void GetMinThreads (out int workerThreads, out int completionPortThreads);  
  
public static void GetAvailableThreads (out int workerThreads, out int completionPortThreads);
```

The APIs take two parameters named `workerThreads` and `completionPortThreads`, which represent the worker threads and asynchronous I/O completion

port threads, respectively. Notice that the return type of these methods is void, and the out parameters are used to populate the result.

The `GetMaxThreads` returns the maximum number of workers and I/O threads allowed in the pool. This API can be used to determine the number of threads in the pool while diagnosing the issues.

The `GetMinThreads` returns the minimum number of workers and I/O threads allowed in the pool. It is the number of threads that the `ThreadPool` creates on-demand as new requests are queued to `ThreadPool`. After these minimum threads are created, `ThreadPool` starts throttling the thread creation process by using its thread management algorithm.

The `GetAvailableThreads`, as the name suggests, returns the available worker and I/O threads in the pool.

Mathematically, *Available Threads = Max Allowed Threads - Threads currently in use.*

So, the available worker threads mean that many more worker threads can be started without throttling. If there are no available threads, additional requests queued to the `ThreadPool` would remain queued until another thread finishes its work and return to pool or every thread takes more than 500 milliseconds, post which `ThreadPool` may create a new thread.

The code sample to get the `ThreadPool` statistics is shown below:

```
static void Main(string[] args)
{
    DisplayThreadStats();
    Console.ReadLine();
}

static void DisplayThreadStats()
{
    int workerThreads;
    int iocompletionThreads;

    ThreadPool.GetMinThreads(out workerThreads, out
    iocompletionThreads);

    Console.WriteLine($"Minimum worker threads are {workerThreads}.

{Environment.NewLine}Minimum I/O Completion threads are
{iocompletionThreads}.{Environment.NewLine}Processor count is
{Environment.ProcessorCount}{Environment.NewLine}");

    Console.WriteLine();

    ThreadPool.GetMaxThreads(out workerThreads, out
```

```
iocompletionThreads);  
    Console.WriteLine($"Maximum worker threads are {workerThreads}.";  
{Environment.NewLine}Maximum I/O Completion threads are  
{iocompletionThreads}.{Environment.NewLine});  
    Console.WriteLine();  
    ThreadPool.GetAvailableThreads(out workerThreads, out  
iocompletionThreads);  
    Console.WriteLine($"Available worker threads are {workerThreads}.";  
{Environment.NewLine}Available I/O Completion threads are  
{iocompletionThreads}.");  
}
```

The output of this program in my machine looks like:

```
Minimum worker threads are 4.  
Minimum I/O Completion threads are 4.  
Processor count is 4  
Maximum worker threads are 32767.  
Maximum I/O Completion threads are 1000.  
Available worker threads are 32767.  
Available I/O Completion threads are 1000.
```

We notice that default minimum worker threads and I/O threads are 4, which is the same as that of the processor count in the machine in which this program was run.

To set the number of minimum or maximum threads, we have the following APIs:

```
public static bool SetMinThreads (int workerThreads, int  
completionPortThreads);  
public static bool SetMaxThreads (int workerThreads, int  
completionPortThreads);
```

I would highly discourage the usage of these APIs unless you totally and completely understand why you need to use them.

The `SetMinThreads` API sets the minimum number of workers and I/O completion port threads. By default, the minimum value of these threads is set to the number of processors in the system. `ThreadPool` creates a minimum number of threads on-demand, and minimum numbers of threads would exist in the `ThreadPool`. Post this, `ThreadPool` thread management algorithm takes over, which throttles the creation of new threads. Even if there are several requests queued afterward, `ThreadPool` would roughly create one thread every 500 milliseconds or so. So, if the requests

queued are less than the minimum number of threads, `ThreadPool` will process it very quickly. However, if requests queued are more than the minimum number of threads, there would be some delay in the processing of requests as `ThreadPool`. `ThreadPool` would either:

1. Wait for an existing thread to get free, so that it can assign the new request to the thread returning to the pool
2. See if no existing thread becomes free in 500 milliseconds, so would create a new thread and assign the request to this newly created thread

Unnecessarily increasing or decreasing the minimum number of threads may hurt the performance

In the chapter on debugging, we will see how we can identify and fix a situation called **thread starvation**, which can be caused due to a lack of available threads in the `ThreadPool`, and we receive a burst of requests.

Exceptions in `ThreadPool`

Unhandled exceptions in the application terminate the process. Unhandled exceptions in methods executed by `ThreadPool` threads are no different, so exception handling should be done as we saw earlier in this chapter. However, below exceptions, doesn't result in termination of the process:

- When a thread is aborted by calling the `Abort` method, `ThreadAbortException` is thrown
- Thread is terminated by the host process

All in all, exception handling should be done correctly in every method that we write. In the next section, we will discuss the limitations of `ThreadPool`.

Limitations of Thread Pool

Though using `ThreadPool` is super easy, it does have a few limitations like:

- Using `ThreadPool`, we cannot change the state or priority of the thread.
- Unlike manually created threads, we cannot assign an identifier to a `ThreadPool` thread and track it.
- `ThreadPool` cannot be used to run multiple tasks in a deterministic fashion as the sequence in which the threads schedule and execute the request depends on the `ThreadPool` scheduler.
- When a request is queued to the `ThreadPool`, depending upon the load and number of available threads, it may execute non-deterministically sometime in the future. If the load is high and there is no availability of threads, it may take longer to execute the request.

- ThreadPool is designed for numerous small requests, so if we need to execute a long-running task, it should be executed on a manually created thread and not on ThreadPool as it can disturb the auto-adjusting thread management algorithm of ThreadPool.
- There is no inbuilt mechanism to know if the request scheduled on a ThreadPool thread has completed.
- Since enqueueing request on ThreadPool accepts a delegate which has a return type of void, we do not have the flexibility to get the return value once the request execution completes.
- While there are few limitations of ThreadPool, it is used extensively by the .NET Core framework internally, as we discussed earlier. Let's see ThreadPool in action in the next section.

ThreadPool in action

Few of the .NET /.NET Core constructs which make use of ThreadPool directly or indirectly are as follows:

- Asynchronous delegates like `Delegate.BeginInvoke`
- `BackgroundWorker` (now mostly obsolete)
- `System.Timers.Timer` and `System.Threading.Timer` and timers derived from them
- Tasks and **Task Parallel Library**, a.k.a **TPL**

The first two are almost obsolete in the modern threading paradigm, and timers are discussed in the upcoming chapter. Let's discuss Task, which makes it even easier to make use of threading or running code in parallel.

Task

To overcome the limitations of ThreadPool, explicitly getting the return value, know when the request execution is completed, the ability to cancel a request), and the ability to support more features, Microsoft introduced the concept of Task. Task represents an asynchronous operation, which implements the **Promise Model of Concurrency**; that is, they represent a non-finished operation but promise that operation will be completed sometime in the future and provide an API to interact with the promise. Task represents an operation that does not have a return value.

The `System.Threading.Tasks` namespace contains the API to leverage the Task:

```
ThreadPool.QueueUserWorkItem(WriteToConsole, "ThreadPoolThread");
```

This code runs the method `WriteToConsole` on the `ThreadPool` thread. The same method can be executed in a new thread via Task API. The following code snippet demonstrates how the Task API can be used to execute an action:

```
static void Main(string[] args)
{
    Console.WriteLine("Task Sample 1");
    //// Task Sample 1 - Create a task only. Don't start
    Task task1 = new Task(WriteToConsole, "ThreadPoolThread");
    Console.WriteLine("Task Sample 2");
    //// Task Sample 2 - Create a started task and wait for it
    to complete.

    Task task2 = Task.Run(() =>
    WriteToConsole("ThreadPoolThread"));

    task2.Wait(); //// Overload exists to specify the timeout.
    //// Start task 1 and wait for it to complete.

    task1.Start();
    task1.Wait();
    Console.WriteLine("Task Sample 3");
    //// Task Sample 3 - Create a started task and wait for it
    to complete.

    Task task3 = Task.Factory.StartNew(() =>
    WriteToConsole("ThreadPoolThread"));

    task3.Wait();
    Console.WriteLine("Task Sample 4");
    //// Task Sample 4 - Create a task and run it synchronously
    on main thread and wait for it to completely.

    Task task4 = new Task(WriteToConsole, "ThreadPoolThread");
    task4.RunSynchronously();
    task4.Wait();
    Console.ReadLine();
}
```

The comments make the code self-explanatory. The sample shows four ways of executing the Task. Few of the key learnings are listed below:

1. Using the new `Task(Action action, object? state)` constructor of `Task` creates a task but doesn't start it. It returns an instance of the created `Task`.

2. The Task can be started by calling the instance method `Start`.
3. We can wait for the task to complete by calling its instance method `Wait`.
4. The `static` method `Run` enqueues the action to the thread pool and returns the instance of the created task. Many people prefer this way of initiating a Task as it creates and starts a task in one line, unlike the new Task constructors, where first the Task is created and then has to be explicitly started.
5. Task has a `static` property named `Factory`, which is of type `TaskFactory` and has a method `StartNew` that takes a delegate as the input parameter and returns an instance of created task.
6. A task can be made to run synchronously on the `Main Thread` using the instance method `RunSynchronously()`.
7. Apart from `task4`, all the other three tasks are executed on the `ThreadPool` thread, and this can be confirmed by checking the `IsBackground` and `IsThreadPoolThread` property printed by these tasks.

In the **Universal Windows Platform (UWP)**, there is no way to create a thread or schedule a request to `ThreadPool` thread as Microsoft deliberately didn't expose these APIs in the UWP. The only way to create a thread in UWP is via `Task`. UWP gives a good indication that Microsoft wants developers to embrace and use `Task`.

The `Task` class is defined in the framework as:

```
public class Task : IAsyncResult, IDisposable
{
    public Task(Action action);
    public Task(Action action, CancellationToken cancellationToken);
    public Task(Action action, TaskCreationOptions creationOptions);
    public Task([NullableAttribute(new[] { 1, 2 })] Action<object?>
action, object? state);

    public Task(Action action, CancellationToken cancellationToken,
TaskCreationOptions creationOptions);

    public Task([NullableAttribute(new[] { 1, 2 })] Action<object?>
action, object? state, CancellationToken? cancellationToken);

    public Task([NullableAttribute(new[] { 1, 2 })] Action<object?>
action, object? state, TaskCreationOptions? creationOptions);

    public Task([NullableAttribute(new[] { 1, 2 })] Action<object?>
action, object? state, CancellationToken? cancellationToken,
TaskCreationOptions? creationOptions);

    //// Code not shown for brevity and focus.
}
```

All the constructors shown above takes an `Action` or `Action<object?>` as a parameter, which means `Task` represents an operation that doesn't return value and executes asynchronously (as `Action` and `Action<T>` doesn't have a return type). Apart from this, the overloads of constructors have `TaskCreationOptions` and `CancellationToken` as the parameters. The `Task` is the central component of the **Task-based Asynchronous Pattern (TAP)**, which we will discuss in detail in *Chapter 6*. `Task` internally uses `ThreadPool` thread to execute the request, which, as we discussed earlier, is optimized for numerous small running requests and not for long-running requests. To create a task executing long-running requests, we would use the overload of `Task` accepting `TaskCreationOptions` enum, as shown below:

```
Task task = new Task(LongRunningMethod, TaskCreationOptions.  
LongRunning);
```

Apart from the long-running tasks, there are other uses of `TaskCreationOptions` enum as well, let's see this enum.

TaskCreationOptions

`TaskCreationOptions` is an enum and is defined in `System.Threading.Tasks` namespace as listed below:

```
[Flags]  
public enum TaskCreationOptions  
{  
    // Specifies that the default behavior should be used.  
    None = 0,  
  
    // Summary:  
    // A hint to a System.Threading.Tasks.TaskScheduler to  
    // schedule a task in as fair  
    // a manner as possible, meaning that tasks scheduled  
    // sooner will be more likely  
    // to be run sooner, and tasks scheduled later will be  
    // more likely to be run later.  
    PreferFairness = 1,  
  
    // Summary:  
    // Specifies that a task will be a long-running, coarse-  
    // grained operation involving
```

```
// fewer, larger components than fine-grained systems. It
provides a hint to the

    // System.Threading.Tasks.TaskScheduler that
    oversubscription may be warranted.

    // Oversubscription lets you create more threads than
    the available number of hardware

    // threads. It also provides a hint to the task scheduler
    that an additional thread

    // might be required for the task so that it does not
    block the forward progress

    // of other threads or work items on the local thread-pool
    queue.

    LongRunning = 2,

    // Summary:
    // Specifies that a task is attached to a parent in the
    task hierarchy. By default,
    // a child task (that is, an inner task created by an
    outer task) executes independently
    // of its parent. You can use the System.Threading.Tasks.
    TaskContinuationOptions.AttachedToParent
    // option so that the parent and child tasks are
    synchronized. Note that if a parent
    // task is configured with the System.Threading.Tasks.
    TaskCreationOptions.DenyChildAttach
    // option, the System.Threading.Tasks.TaskCreationOptions.
    AttachedToParent option
    // in the child task has no effect, and the child task
    will execute as a detached
    // child task. For more information, see Attached and
    Detached Child Tasks.

    AttachedToParent = 4,

    // Summary:
    // Specifies that any child task that attempts to execute
    as an attached child task
    // (that is, it is created with the System.Threading.
    Tasks.TaskCreationOptions.AttachedToParent
```

```
// option) will not be able to attach to the parent task  
and will execute instead  
// as a detached child task. For more information, see  
Attached and Detached Child Tasks.  
DenyChildAttach = 8,  
  
// Summary:  
// Prevents the ambient scheduler from being seen as the  
current scheduler in the  
// created task. This means that operations like StartNew  
or ContinueWith that are  
// performed in the created task will see System.  
Threading.Tasks.TaskScheduler.Default  
// as the current scheduler.  
HideScheduler = 16,  
  
// Summary:  
// Forces continuations added to the current task to be  
executed asynchronously.  
RunContinuationsAsynchronously = 64  
}
```

The comments make it comprehensive to understand what each item in the enum stands for. Notice the attribute `Flags` on the enumeration so that we can bitwise-OR multiple values together. It's good to know that the task can be created for a variety of cases by making use of `TaskCreationOptions` enum. When we pass the flags to the constructor, it's like telling the `TaskScheduler` what kind of task to create, but it's up to the `TaskScheduler` whether to honor the flags or not.

So far, we have just executed the `Action<object>` via `Task`, which doesn't have a return type. However, there is `Task<TResult>` class, derived from `Task`, which can be used to execute a `Func<T, TResult>`, which has a return type of `TResult`. It is defined as shown below:

```
// Summary: Represents an asynchronous operation that can return  
a value.  
// Type parameters:  
// TResult: The type of the result produced by this System.  
Threading.Tasks.Task`1.  
public class Task<TResult> : Task
```

The return value is stored in the `Result` property of the `Task`. Calling `task.Result` would block until the task finishes, just like the `task.Wait()`:

Let's see how we can use it:

```
static void Main(string[] args)
{
    int number = 1000;
    Task<int> task = new Task<int>(SumOfNumbers, number);
    task.Start();
    task.Wait();
    Console.WriteLine($"The sum of first {number} is {task.Result}");
    Console.ReadLine();
}

/// <summary>
/// Calculates the sum of numbers from 0 to the specified number.
/// </summary>
/// <param name="numberTo">The number till which sum needs to be
calculated.</param>
/// <returns>The integer number that is sum</returns>
static int SumOfNumbers(object numberTo)
{
    var num = (int)numberTo;
    int sum = 0;
    if (num > 0)
    {
        for (int i = 0; i <= num; i++)
        {
            sum += i;
        }
    }

    return sum;
}
```

Sometimes we have a result available already, but we need to make a task return it. Task API facilitates this scenario as well via `Task.FromResult()`. This scenario is especially useful in unit testing, wherein we may need to mock-up a task returned by a function.

Task has much better and comprehensive support for exception handling as well. Let's see a sample to understand it.

Exception handling with Tasks

If any exceptions are thrown from within a task, they will be wrapped in an `AggregateException`.

To simulate an exception, we throw an exception from the method and invoke it via Task API. We follow the familiar `try...catch` block and catch `AggregateException`. `AggregateException`, as the name suggests, represents one or more errors that occur during application execution. It has an instance method `Handle` defined as:

```
public void Handle (Func<Exception,bool> predicate);
```

It accepts a predicate with exception as an input and returns a Boolean representing if the exception was handled. It invokes a handler callback on each exception contained in the `AggregateException`, and the method can decide how to handle the exception and return `true` if handled or `false` if unhandled. If at the end of the `Handle` method, there is an exception that is not handled, then a new `AggregateException` is created containing the unhandled exceptions, and this new `AggregateException` is thrown. Let's see a sample demonstrating exception handling with Task:

```
static void Main(string[] args)
{
    Task<int> task = new Task<int>(MethodThatThrowsException,
null);
    try
    {
        task.Start();
        Console.WriteLine($"The result is {task.Result}"); // 
task.Wait() would have the same effect.
    }
    catch (AggregateException ex)
    {
        ex.Handle((e) =>
```

```

    {
        if (e is InvalidOperationException)
        {
            Console.WriteLine($"Caught Exception - {e.Message}");
            return true;
        }
        return false;
    });
}

Console.ReadLine();
}

static int MethodThatThrowsException(object state)
{
    throw new InvalidOperationException("Throwing Exception for demonstrating Exception handling in Tasks");
}

```

We notice from this example that, unlike threads, unhandled exceptions thrown from method running inside a task are propagated back to the calling thread. Exceptions are propagated when we use `task.Wait()` or `task.Result`, and you handle them by enclosing the call in a `try..catch` statement. These exceptions are precisely the reason that in the above sample, we enclose the `task.Result` in the `try` block. We use the `Handle` method to handle the exception. Alternatively, we can also use a `foreach` loop to iterate over the `InnerExceptions` property of `AggregateException` or use the instance method `Flatten` and iterate over `InnerExceptions`.

Note on exception handling: If we do not call `Wait()` or `Result` or query exception property of the task, then the exception even if thrown is never observed by our code which is not a good scenario to be caught in if something goes wrong, and we as a developer of code doesn't have a clue about it. Such exceptions can be caught by `TaskScheduler's static UnobservedTaskException event`. The argument of this event contains the `AggregateException`.

One of the exceptions, which I observed most frequently in the Web applications during high load, is `TaskCancelledException`. This exception generally occurs in case of time outs when the Task fails to complete within the specified time. It brings

us to another critical user scenario supported by `Task`, cancellation.

Cancellation

Often we encounter a situation where we need to cancel a long-running operation; for example, it happens with me that I make some changes in code and trigger a build in Visual Studio and then realize that I need to make one more change. Fortunately, Visual Studio offers the option to cancel the build. Likewise, cancellation is desired in many software scenarios as well. `Task`, however, provides an overall pattern to cancel the requests. The pattern is generally termed as cooperative cancellation as it needs the method to support cancellation if you want to cancel it. Such methods can be easily identified as they would take `CancellationToken` as one of the parameters. To enable cancellation support, we need to do the following:

1. Create an object of type `System.Threading.CancellationTokenSource`. This class is defined as:

```
[ComVisible(false)]
public class CancellationTokenSource : IDisposable
{
    public CancellationTokenSource();
    public CancellationTokenSource(TimeSpan delay);
    public CancellationTokenSource(int millisecondsDelay);
    public bool IsCancellationRequested { get; }
    public CancellationToken Token { get; }
    public void Cancel();
    public void Cancel(bool throwOnFirstException);
    public void CancelAfter(TimeSpan delay);
    public void CancelAfter(int millisecondsDelay);
    //// Other members not shown for brevity.
}
```

Apart from the constructors, there is a Boolean property `IsCancellationRequested` which is set to `true` if cancellation is requested by calling one of the overloads of the `Cancel` method or `CancelAfter` methods. Another valuable property is the `Token` of type `CancellationToken` is passed to the methods that need to support cancellation. `CancellationToken` is a value type and defined as:

```
public struct CancellationToken
{
```

```
        public CancellationToken(bool canceled);
        public static CancellationToken None { get; }
        public bool IsCancellationRequested { get; }
        public bool CanBeCanceled { get; }
        public void ThrowIfCancellationRequested();
        //// Other members not shown for brevity.
    }
```

2. `CancellationToken` is a struct. It has a Boolean property `IsCancellationRequested` which the method supporting cancellation can check periodically (in a loop) and decide if it needs to exit the loop or operation.
3. Pass the `Token` property associated with the `CancellationTokenSource` to the method and task overload.
4. Check the `IsCancellationRequested` property of the `CancellationToken` token and if it's true, break and exit the method or call `ThrowIfCancellationRequested`, which would throw an `OperationCancelledException` and exit the method.
5. Handle the `AggregateException` in the catch block.

The simple cancellation sample is shown below, which requests cancellation by calling `CancelAfter` method:

```
static void Main(string[] args)
{
    // Initialize a cancellation token source
    var cts = new CancellationTokenSource();

    // Run the MethodThatCanBeCancelled passing in the token
    // from cancellation token source and keep its reference.

    Task t = Task.Run(() => MethodThatCanBeCancelled(1000000,
        cts.Token), cts.Token);

    try
    {
        // Cancel after 1000 ms.
        cts.CancelAfter(1000);

        // Till then, wait for task. Unless we wait, exception
        // if any can't be caught.
    }
```

```
        t.Wait();
    }
    catch (AggregateException ex)
    {
        // Catch Aggregate exception and call its handle method.
        ex.Handle((e) =>
    {
        // Check if exception is operation canceled exception.
        if (e is OperationCanceledException)
        {
            // Log the exception message to Console.
            Console.WriteLine($"Operation Cancelled!
{e.Message}");
            // return true, signifying that exception is
handled.
            return true;
        }

        // Its not operation canceled, bubble up, by
returning false.
        return false;
    });
}
finally
{
    // Finally dispose the cancellation token source.
    cts.Dispose();
}

Console.ReadLine();
}

/// <summary>
/// Defines a method that can be cancelled.
```

```
/// </summary>
/// <param name="state">The object state.</param>
/// <param name="token">The cancellation token.</param>
static void MethodThatCanBeCancelled(object state,
CancellationToken token)
{
    // Check if cancellation is requested.
    if (token.IsCancellationRequested)
    {
        // Yes, cancellation is requested. Write this status on
        // the console.
        Console.WriteLine($"Cancellation requested even before
it started...");

        // Throw cancellation exception
        token.ThrowIfCancellationRequested();
    }

    // Cast state to int as we pass integer in the caller
    // method.
    int length = (int)state;

    // Run the loop
    for (int i = 0; i < length; i++)
    {
        // Simulate the work that runs for a while.
        Task.Delay(5000);
        if (token.IsCancellationRequested)
        {
            // Yes, cancellation is requested. Write this status
            // on the console.
            Console.WriteLine($"Cancellation requested while
running iteration # {i}");

            // Throw cancellation exception
            token.ThrowIfCancellationRequested();
        }
    }
}
```

```
    }  
}
```

So far, in all the samples, we saw that we create a task and then `Wait()` for it. Though this is fine for a sample program to demonstrate the usage, it would not be a good idea to follow the same pattern in enterprise software as it would result in the blocking of threads. So, it's desired that as soon as a task finishes, we should be able to process it or continue with another task immediately without blocking the threads. This can be achieved by Task continuations, which we will discuss in the next section.

Continuations

Task continuations can be achieved easily by using `ContinueWith` construct. `ContinueWith` will ensure that as soon as the previous task completes, the new task can start immediately on the thread pool. `ContinueWith` returns the reference of the new task, and what's more, we can call several `ContinueWith` from the same task object. When the task completes, all the `ContinueWith` would be queued to the `ThreadPool`. `ContinueWith` has an overload that takes the `TaskContinuationOptions` enum as the parameter which has very identical values to the `TaskCreationOptions` we discussed earlier and more values as well, which can be bitwise -OR-ed as needed. The `TaskContinuationOptions` enum is defined as:

```
[Flags]  
public enum TaskContinuationOptions  
{  
    None = 0,  
    PreferFairness = 1,  
    LongRunning = 2,  
    AttachedToParent = 4,  
    DenyChildAttach = 8,  
    HideScheduler = 16,  
    LazyCancellation = 32,  
    RunContinuationsAsynchronously = 64,  
    NotOnRanToCompletion = 65536,  
    NotOnFaulted = 131072,  
    OnlyOnCanceled = 196608,  
    NotOnCanceled = 262144,  
    OnlyOnFaulted = 327680,
```

```
    OnlyOnRanToCompletion = 393216,  
    ExecuteSynchronously = 524288  
}
```

This enum makes the continuation very flexible, and we can continue the tasks only in exceptional cases like in case of exception (`TaskContinuationOptions.OnlyOnFaulted`) or cancellation (`TaskContinuationOptions.OnlyOnCanceled`). By default, if you do not specify any of these flags, then the new task will run regardless of how the first task completes. Here is a simple sample that demonstrates continuations:

In this sample, we simulate a task that fetches data, and then as soon as it finishes, it continues with the processing of data and post that displays the data. Notice that the result of the previous task can be used inside `ContinueWith`:

```
static void Main(string[] args)  
{  
    Task<int> t = new Task<int>(GetData, null);  
    t.Start();  
    t.ContinueWith(previousTaskResult =>  
        ProcessData(previousTaskResult.Result)).ContinueWith(lastTaskResult =>  
        Console.WriteLine($"Displaying the processed data as {lastTaskResult.  
        Result}"));  
    Console.ReadLine();  
}  
  
static int GetData(object state)  
{  
    Console.WriteLine("Getting Data starts");  
    // Simulate Getting data from DB or from API from the network.  
    Task.Delay(3000);  
    Console.WriteLine("Getting Data ends");  
    Console.WriteLine("-----");  
    return 23101506;  
}  
static int ProcessData(object state)  
{  
    Console.WriteLine("Process Data starts");
```

```
        var input = (int)state;
        Console.WriteLine($"Received previous task result as {input}");
        // processing of data.
        var result = 2 * input + 1;
        Console.WriteLine($"The result of data processing is {result}");
        Console.WriteLine("-----");
        return result;
    }
```

The previous code listing illustrates usage of task continuations. However, we saw only a single task and its continuation. In real enterprise applications, we run into cases, where in a task needs to be performed when one or all the previously running tasks are completed. We will see how we can handle such cases, in the next section.

WhenAll, WhenAny

At times we run into a situation, wherein we have multiple tasks running in parallel, and we need to take the next step when either:

1. All the tasks are completed, or
2. Any of the tasks are completed

To handle such scenarios, Task class offers static methods `WhenAll`, `WhenAll<T>`, and `WhenAny`, `WhenAny<T>`. There are `WaitAll`, `WaitAll<T>` and `WaitAny`, `WaitAny<T>` as well, but since these are blocking, I do not recommend using them. In this discussion, we will only see `WhenAll`, and the rest of these can be similarly used. Following are the commonly used overloads of `WhenAll`:

```
public static Task WhenAll(IEnumerable<Task> tasks);
public static Task<TResult[]> WhenAll<TResult>(params Task<TResult>[] tasks);
public static Task<TResult[]>
WhenAll<TResult>(IEnumerable<Task<TResult>> tasks);
public static Task WhenAll(params Task[] tasks);
```

We have overloads to handle the methods returning values as well as returning `void`. We have two overloads of each, one making use of `params` and other taking `IEnumerable`. The return type of `WhenAll` is a `Task` object which would execute when all the tasks specified in the `WhenAll` completes. It comes with its pros and cons. If any of the task faults, then this task would also end up being in a faulted state. The exception would be wrapped in the `AggregateException` of the task, which can be caught upon waiting or querying the `Result` property. If any of the

tasks don't fault, but one of them is canceled, then this task also ends up being in the canceled state. Suppose we have three tasks, `t1`, `t2`, `t3`, and we wish to continue processing when all these tasks are done.

We can achieve this by the following:

```
var t4 = Task.WhenAll(t1, t2, t3);
try
{
    t4.Wait();
    // Continue other tasks.
}
catch (AggregateException ex)
{
    // Handle exception by calling ex.Handle
}
```

On similar lines, `WhenAny`, `WaitAny`, `WaitAll` can be used. It is left as an exercise for the reader.

Task Scheduler

Task infrastructure provides us great flexibility and overcomes the limitations offered by `Thread` and `ThreadPool` constructs. We discussed earlier in the chapter that when we start a task, it gets scheduled into the `ThreadPool` thread. Who does this? Task Scheduler.

`TaskScheduler` is an abstract class defined in `System.Threading.Tasks` namespace as shown below:

```
//Represents an object that handles the low-level work of
queueing tasks onto threads.

[DebuggerDisplay("Id={Id}")]
[DebuggerTypeProxy(typeof(System.Threading.Tasks_
TaskSchedulerDebugView))]
public abstract class TaskScheduler
{
    protected TaskScheduler();
    public static TaskScheduler Default { get; }
    public static TaskScheduler Current { get; }
```

```
    public virtual int MaximumConcurrencyLevel { get; }

    public int Id { get; }

    public static event
EventHandler<UnobservedTaskExceptionEventArgs> UnobservedTaskException;

    public static TaskScheduler
FromCurrentSynchronizationContext();

    protected abstract IEnumerable<Task> GetScheduledTasks();

    protected bool TryExecuteTask(Task task);

    protected abstract bool TryExecuteTaskInline(Task task, bool
taskWasPreviouslyQueued);

    protected internal abstract void QueueTask(Task task);

    protected internal virtual bool TryDequeue(Task task);

}
```

`TaskScheduler` also exposes task-related information to the debugger. The framework offers two types of `TaskScheduler`:

1. `ThreadPool` task scheduler
2. `SynchronizationContext` task scheduler

By default, all applications use the `ThreadPool` task scheduler, which schedules the tasks to the worker threads of `ThreadPool`. The `SynchronizationContext` task scheduler is mostly used by **Graphical User Interface (GUI)** based application like Windows Forms, WPF, and many more. The purpose of this scheduler is to schedule the task to the UI thread so that the task can update the UI components. Recall that in Windows Forms, WPF applications, only the thread creating the UI controls can update the UI; that is, UI thread. `SynchronizationContext` task scheduler doesn't schedule a task to `ThreadPool` worker threads but to UI threads. We can provide a reference to the `SynchronizationContext` task scheduler by using the `TaskScheduler`'s static method `FromCurrentSynchronizationContext`.

The `TaskScheduler` class has two `static` properties named `Current` and `Default`, which returns the current `TaskScheduler` instance associated with the `Task` and the default `TaskScheduler` instance provided by the framework. The default `TaskScheduler` is the `ThreadPool` thread scheduler.

It also has an instance property `Id` of type `int`, which gives the unique identifier for the `TaskScheduler`. `UnobservedTaskException` catches any unobserved exceptions that were not explicitly waited or caught by the user code. There are members to:

1. Get the currently scheduled tasks. It is primarily for debuggers.

2. Queue the task to the scheduler.
3. Dequeue the task previously scheduled.

Following is a sample to schedule a task on ThreadPool thread and UI thread:

```
CancellationTokenSource cts = new CancellationTokenSource();
// Execute task on ThreadPool Thread (Default)
Task<int> task = Task.Run(() => GetData(), cts.Token);
// Execute task on the UI thread (generally used to update UI)
task.ContinueWith(t =>
{
    // Update data in UI, like updating a label in WPF/
    Windows Form application.
}, cts.Token, TaskContinuationOptions.NotOnCanceled,
TaskScheduler.FromCurrentSynchronizationContext());
```

The Task can be scheduled to the ThreadPool by the TaskScheduler, via a worker thread (a ThreadPool thread) or a non-worker thread. When the task is scheduled via a non-worker thread, it is added to the global queue (Please refer to the Thread Pool visual representation in ThreadPool section earlier). When the task is scheduled by a worker thread, it is added to the local queue of the worker thread. Whenever the worker thread is free and ready to process operation, it always checks its local queue first. If a task exists, it will dequeue the task from its local queue and process it. This operation is done from the top of the queue in **last-in, first-out (LIFO)** fashion. It is done deliberately to avoid unnecessary synchronization locks and improve performance.

Only the owner worker thread can access the head of the queue to add or remove tasks, so no synchronization lock is needed. It comes at the cost that the last item is processed first; that is why tasks scheduled on ThreadPool are never guaranteed to run in order. If the task doesn't exist in the local queue of the worker thread, it looks in the local queue of the other worker thread and tries to "steal" it from the tail of that queue. Since the tail of the queue can be accessed by multiple worker threads, a synchronization lock is needed, which may hurt the performance a little bit. If the task is not found in the other local queue as well, then the worker thread tries to look at the global queue for tasks and processes it in FIFO fashion, as we discussed earlier. If no tasks exist in the local queue as well as a global queue, then the worker thread would put itself in a sleep state waiting for a task to arrive. If the worker thread remains in a sleep state for a certain amount of time, ThreadPool's thread pool algorithm may decide to destroy this thread based on its logic.

Task Factory

At times, we may need to do a number of similar type of operations that require tasks sharing the same type of configurations, like same `TaskCreationOptions`, `TaskScheduler`, `TaskContinuationOptions`, return type. Rather than keep passing the same parameters over and over again to task constructor for creating new tasks, we can make use of `TaskFactory` to create such tasks, which makes life easier for developers. Here is a quick sample to see it in action:

```
var cts = new CancellationTokenSource();
// Create task factory
var taskFactory = new TaskFactory<int>(cts.Token,
TaskCreationOptions.None,
TaskContinuationOptions.ExecuteSynchronously,
TaskScheduler.Default);
// Use factory to create task.
taskFactory.StartNew(() => M1(cts.Token, "parameter1"));
```

Summary

In this chapter, we discussed the basics of threading. We learned the reason why threading is needed and scenarios where it can be applied. We then learned how to create a thread and get multiple methods execute simultaneously using threads. We saw that threads need careful exception handling and discovered that the creation of threads is expensive. Hence, we should use `ThreadPool`, which manages the threads' lifetime and saves on resource and performance costs associated with manually creating threads. However, we still encountered challenges in knowing when the task completed and getting its result value, so we discussed `Tasks`, which overcomes these challenges. We saw several samples and discussed various `Task` constructs in detail.

Discussion is deliberately done in the order of their evolution in chronological order. Continuing with this philosophy, we will next discuss the `async await` constructs, which is now the recommended pattern for any asynchronous operations. We will discuss this in the next chapter.

Exercise

1. What are the common scenarios in which threading should be used?
2. Differentiate between multithreaded, multi-core, and multiple CPU processors.

3. Identify the number of cores in your computer. Also, list the number of threads running in your machine.
4. What namespace needs to be added to leverage threading?
5. Explain `ThreadLocal` and thread environment block and their use.
6. Create a table with three columns `Threads`, `ThreadPool` & `Tasks`. List the advantages, disadvantages, and limitations of each of them.
7. Which of the following would you prefer to execute a long-running compute-bound operation? Why?
 - a. `Thread`
 - b. `ThreadPool`
 - c. `Task`
8. Explain the work-stealing mechanism used by `ThreadPool`.

CHAPTER 5

Parallel Programming

"A working program is one that has only unobserved bugs."

A tester's perspective.

In the last chapter, we discussed how the effective use of multithreading schedules the tasks into multiple cores, which increases the throughput. However, Task-based constructs may seem overwhelming to most developers. To simplify the usage and adoption, C# came up with a language feature called async-await. In this chapter, we will conclude our discussion on Task Parallel Library (TPL) and then hop on to learn async-await, how to use it and how it works under the hood and the new C# 8 asynchronous streams.

Structure

- Understanding the jargon
- Parallel Extensions
- Task Parallel Library
- Task and Data Parallelism
- Data Structures for parallelism
- IEnumarator and yield return

- async-await
- async-await – Control flow
- async-await – Under the hood.
- Principles for using async-await
- Restrictions in using async-await
- Deadlock
- Asynchronous streams
- ValueTask
- Summary
- Exercise

Objectives

By the end of this chapter, the reader should be able to understand:

- The terms like concurrency, parallelism, multithreading, asynchrony
- CPU bound and I/O bound operations
- And use an async-await language feature
- How async-await works under the hood
- Convert synchronous method to asynchronous method
- Deadlocks and how to avoid it
- Asynchronous Streams
- ValueTask

Understanding the jargon

When talking about parallel programming, the following terms pop up frequently. They may all appear to be the same but have subtle differences. Let's look at each of these terms:

- **Parallelism:** This is a frequently used term in the world of parallel programming, but often misunderstood and confused with concurrency. Parallelism comes into the picture when at-least two threads are executing simultaneously. It is easily achieved in modern hardware, which has multicore processors and so multiple threads can run simultaneously. It results in better throughput.
- **Concurrency:** When at least two threads are making progress at the same time. But it doesn't mean that they're running simultaneously. One of the examples is ASP.NET Core Web API handling multiple requests. It is

generally achieved by context switching of thread. It results in more work in less time.

- **Asynchrony:** Refers to the absence of synchrony/synchronization. If an operation occurs outside of the main flow, then that operation is referred to as an asynchronous operation. For example, in a WPF application, while the main thread handles the user events on the UI, a background thread fetching the data from Web API, is an asynchronous operation.
- **Multithreading:** It means executing multiple threads concurrently. For example, `ThreadPool` providing multiple threads to handle multiple requests concurrently.

.NET Core is an evolving framework, and with every release, the number of supported features and APIs has increased along with the stability of the framework. In this regard, .NET full framework is a more mature framework. It exposes a few of the best libraries (in my opinion, compared to another language ecosystem like Java) for the developers to perform parallel programming efficiently. One of the libraries that Microsoft came up with almost about a decade back is the Parallel Extensions library.

The focus of this book is .NET Core and C# 8, and not .NET, but for the sake of completeness, we will discuss Parallel Extensions and its components briefly so that the reader has a fair idea about them. I would highly recommend the reader to go through the links and exercise at the end of this chapter, to gain a deep understanding of this topic.

Parallel Extensions

Microsoft is a company that focuses on and invests in research. It's no surprise that most of the innovations that we see in the .NET world have their origin from the research wing of Microsoft called **Microsoft Research (MSR)**. Parallel Extensions was the code name of the project that was the joint venture of the .NET CLR team and MSR. This library came into existence from .NET 4.0 onwards and supported:

- Declarative data parallelism
- Imperative data parallelism
- Imperative task parallelism
- Data structures used for synchronization and execution

The Parallel Extensions library comprises of the following key components:

- **Task Parallel Library (TPL):** For a task as well as data parallelism
- **Parallel LINQ (PLINQ):** For data parallelism
- **System.Collections.Concurrent data structures:** Set of data structures to support, execute and synchronize the parallel and concurrent tasks

Essentially, it is a .NET library, that is, standard .NET code that can be leveraged by any other .NET/.NET Core code. An immediate question that comes to mind is, "What does declarative and imperative prefixes used above mean in this context?". So here it is:

Imperative: Suppose, you are hungry. You ask your cook to prepare rice for you by following the steps like:

1. Take one bowl of rice and wash it with water, until its thoroughly clean.
2. Take a pressure cooker and pour two bowls of water and one bowl of rice to it.
3. Add a pinch of salt and let it cook for three whistles.
4. Let the pressure cooker cool for 10 minutes and then serve the rice.

It is an example of imperative programming, where you instruct what you want and step by step procedure of how you want it. To see a programmatic example, let us discuss a program to find the even numbers in a list of first 100 numbers. The imperative program looks like:

```
IEnumerable<int> numbers = Enumerable.Range(1, 100); ////  
Collection of numbers from 1 to 100  
  
List<int> evenNumbers = new List<int>(); //// Collection to  
store even numbers  
  
foreach (int number in numbers) //// Iterate through all  
numbers  
{  
    if (number % 2 == 0) //// Check if its divisible by 2  
    {  
        evenNumbers.Add(number); //// If yes, add it to the  
even numbers collection.  
    }  
}
```

In the other program, we specify the step by step approach to finding the even numbers in the given collection. This approach is called imperative programming approach where you instruct the compiler to follow the steps that you want and get to the result.

Declarative: Suppose you are hungry again! You ask your cook to prepare some rice for you to eat. The cook prepares the rice and serves it back to you. Sounds simple! It is an example of declarative programming. You just declare the result you want (rice), but do not describe or layout the steps that the cook needs to follow to

prepare the rice (like use one bowl rice, wash in water multiple times, till it is clean. Put it in a pressure cooker. Add a pinch of salt and two bowl water and let it cook for three whistles). The cook is free to use the microwave for cooking or a pressure cooker, or bowl it in just another container to prepare the rice. The steps are not laid out. In declarative programming, you just declare the result that you want, but do not worry about the steps. To see a programmatic example, let us discuss the same problem to find the even numbers in a list of first 100 numbers. The declarative program looks like:

```
IEnumerable<int> numbers = Enumerable.Range(1, 100); /////
Collection of numbers from 1 to 100

IEnumerable<int> evenNumbers = numbers.Where(j => j % 2 ==
0); ///// Get even numbers
```

In the other program, we just specify that give the numbers that are even and do not state that iterate the collection, check divisibility by 2, if divisible, add it to even numbers. This approach is called a declarative programming approach.

In our general programming, we make use of both of these approaches without actually thinking about it being imperative or declarative. So most of the code that the developers write today is a mixed approach or hybrid approach code.

Next, let us discuss each of the components of Parallel Extensions and start with Task Parallel Library.

Task Parallel Library (TPL)

In the last chapter, we had a detailed discussion on Task infrastructure, which is nothing but one of the abstractions available in **Task Parallel Library**, a.k.a **TPL** (used going forward). TPL, along with PLINQ, are two of the most popular libraries for parallel programming in the world of .NET. Recall that Task is an action. It's like a promise (in JavaScript terminology) of operation that would be completed sometime in the future. Generally, we initiate an operation and get a **Task** object for it, which completes when the operation completes. The task may succeed or fail. When it succeeds, it may have the desired result, and in case of failure, it would have the exception or error information. This task represents an asynchronous operation and is more efficient and scalable.

As per MSDN documentation, TPL is a set of public types and APIs in the **System.Threading** and **System.Threading.Tasks** namespaces. The purpose of this library is to make the life of developers easy and more productive by providing more natural and intuitive APIs to develop parallel and concurrent applications.

TIP: To check the .NET managed APIs, use the .NET API browser exposed by Microsoft online at <https://docs.microsoft.com/en-us/dotnet/api/>. It contains all the .NET APIs for full .NET framework or .NET Standard or Xamarin or Azure SDK or ASP.NET Core or .NET Core or ML.NET. It is a convenient way to see the API documentation online and should easily find a way in your browser bookmark bar for quick access to .NET APIs. There is a search and filter feature, which easily lets you search the API you are looking for the selected framework. For .NET Core 3.1, which is the focus of this book, the link is <https://docs.microsoft.com/en-us/dotnet/api/?view=netcore-3.1>.

From the .NET API browser, let us check the APIs and types exposed in the `System.Threading` and `System.Threading.Tasks` namespaces for .NET Core 3.1. You will see a bunch of types and APIs; In this section, we shall be summarizing the most commonly used APIs. However, I would strongly encourage the readers to explore and see the samples of the remaining APIs from the official Microsoft documentation links, which are available in the *Exercise* section of this chapter. A quick summary of everyday operations and their sample code via task is displayed in the next table:

Operation	Pseudo Code
Create a Task	<pre>Task task = Task.Factory.StartNew(() => LongOperation()); Task task = Task.Run(() => LongOperation()); Task task = new Task(() => LongOperation()); task.Start();</pre>
Check Status	<code>task.Status</code>
Check if the canceled	<code>task.IsCancelled</code>
Check if completed	<code>task.IsCompleted</code>
Check if faulted	<code>task.IsFaulted</code>
Check the exception	<code>task.Exception</code>
Check result	<code>task.Result</code> (Not recommended to be used, unless it's the only way)
Wait for task	<code>task.Wait</code> (Not recommended to be used, unless it's the only way)
Compose tasks	<pre>task.ContinueWith (t => {}) Task.WhenAll(t1,t2); where t1, t2 are tasks Task.WhenAny(t1,t2); where t1, t2 are tasks Task.WaitAll(t1,t2); (Not recommended, use WhenAll() instead) Task.WaitAny(t1,t2); (Not recommended, use WhenAny() instead)</pre>

Table 5.1: Task usage

`LongOperation` is a synchronous method with the following signature and does a long compute-bound operation:

```
void LongOperation()
{
    //// Long running code
}
```

Let us dive into data and task parallelism next.

Data parallelism

Apart from `Task`, another type of interest is the static `System.Threading.Tasks.Parallel` class, which provides the parallel counterparts for the most common scenarios encountered in the real world programming. Most developers have to write `for` and `foreach` loop in their day to day code while working with collections. Both these loops run sequentially. For a small number of iterations, it is generally fine. However, if the number of iterations is high, running these sequential loops is time-consuming and doesn't make efficient use of multiple CPU cores available to us. The `Parallel` class provides the goodness of parallelism in the `for` and `foreach` loop that we can leverage in such scenarios to utilize the CPU cores optimally. Let us see how:

Consider the following code:

```
void CheckEvenAndWrite(int number)
{
    if (number % 2 == 0)
    {
        Console.WriteLine($"{number} is even");
    }
    else
    {
        Console.WriteLine($"{number} is odd");
    }
}
```

We have a method, which takes an integer as input. It checks if the number is even or odd and then writes it to console. Though it doesn't do anything of significance, this code snippet intends to show that this method is dependent on input (data) to show if it's even or odd. This method need not be called in any given sequence to work correctly.

If the above method is called for the first 1000000 numbers, it will take a considerable amount of time to print the odd and even status for each of the numbers:

```
for (int i = 0; i < 1000000; i++)
{
    CheckEvenAndWrite(i);
}
```

In the above case, one thread (the main thread in case of Console app since we are writing to console) would be executing this loop and that too sequentially. Suppose execution for 1 number takes 1ms time. Then for 100000 numbers, it would take 100 seconds! Though this number is hypothetical and each iteration may take the time of the order of a few microseconds, the point here is that it is sequential, executed by a single thread. As the iteration count increases, time to execute would also increase linearly, and it doesn't utilize the multiple CPU cores that are available in all modern computers and servers today.

With the `Parallel` construct of `for` loop, we can pretty much solve the above issues, without much code change. The parallelized code for the above method, using a `static Parallel` class is as shown below:

```
//// ThreadPool threads would execute this in parallel.
Parallel.For(0, 100000, i => CheckEvenAndWrite(i));
```

When we run above two codes (sequential and parallel) and compare, we will find that in a multicore machine (which is not fully loaded already):

1. Time taken by sequential code is higher than the parallel counterpart.
2. If you open the Resource monitor in the Performance tab of your Task Manager (*Windows + R | Run taskmgr | Open Resource monitor*), you will notice that while running the parallel code, all the cores are utilized. In contrast, in the sequential code, only one Core is utilized. It is, of course, the case if you are working on a multicore machine.
3. The critical thing to notice is that, in the sequential code, all the numbers would be printed in sequence, while in parallel code, the numbers are out of sequence, so this should be your very first consideration that if you need the processing to happen in sequential order, do not use parallel counterpart.

Important note: The parallel construct is not a silver bullet to handle all your performance issues. You must carefully benchmark your code using both sequential and parallel constructs, and only if you get a significant enhancement in performance, you should promote your code to a higher environment. Wrongly using parallel constructs can lead to performance degradation. In a few cases, the wrong parallelization may lead to hard to find bugs! So, be very careful and alert in your unit testing, whenever you make use of parallel constructs.

Just like `for` loop, we can do similar operations on the `foreach` loop as well. Suppose instead of iterating from 1 to 100000; we have the collection of a number as shown in the following code:

```
IEnumerable<int> numbers = Enumerable.Range(1, 100000);
```

Consider the `foreach` loop shown below:

```
foreach(int number in numbers)
{
    CheckEvenAndWrite(number);
}
```

This `foreach` loop has a collection (`numbers`), a loop variable (`number`), and a loop body (code with parenthesis).

The `Parallel` counterpart of the above `foreach` loop is simple, as shown below, with minimal code changes:

```
Parallel.ForEach(numbers, number =>
{
    CheckEvenAndWrite(number);
});
```

It also has a collection (`numbers`), a loop variable (`number`), and a loop body (code within parentheses). The above code is parallel but not asynchronous. If you have a flexibility that you can use either `For` or `ForEach`, then you should choose `For`, as `For` executes much faster.

In the preceding example, we as programmers define how we process the data by iterating the collection in a step by step manner. Therefore, the above samples are examples of imperative data parallelism. In these samples, data is partitioned and used by multiple threads that operate on a different segment of data concurrently.

What happens if there are one toy and three children wanting to play with the same toy? A fight, contention, yes! It is the same thing that happens when we introduce parallelism and try to modify a shared resource inside the loop. So please be extra careful while trying to introduce `Parallel` constructs in your code that uses shared resources or data.

Exceptions are part and parcel of code and may happen anytime. Any exception that occurs inside the parallel loops would be thrown as `AggregateException` that we discussed earlier. It is because, behind the scenes, parallel constructs make use of tasks, which throws `AggregateException` in case of exceptions. So, while doing

exception handling ensure, you catch `AggregateException`. Suppose in our parallel loop, four tasks are executing concurrently, and due to some wrong code or some other environmental issue, the exception occurs in one of the tasks. What would happen? Does it kill the executing tasks? No. It doesn't kill any executing tasks, which are already "in-flight." An error would be captured, and all the executing tasks would finish. Then, all the exceptions would be aggregated and thrown. Post this, no new tasks will be started.

A keen reader would soon think that since `Parallel` constructs utilize all the cores of CPU, can it be the case that my CPU gets saturated? Well! The answer to this question is, Yes, it's possible. Then, how can we get around this scenario? We have a class called `ParallelOptions`, where we can limit the CPU usage by setting the `MaxDegreeOfParallelism` property. By default, its value is equal to the number of CPU cores (`Environment.ProcessorCount`). Depending upon our requirements, we can set `MaxDegreeOfParallelism` to a value derived after conducting a performance benchmarking of our code. The cancellation support in parallel loops is also provided by the `ParallelOptions` class. There is another property called `TaskScheduler`, which can be set to specify which `TaskScheduler` should be used. Setting this property to null would use the default task scheduler. Following is the code for the `ParallelOptions` class as defined in the framework:

```
public class ParallelOptions
{
    // Gets or sets the System.Threading.CancellationToken
    // associated with this System.Threading.Tasks.ParallelOptions
    // instance.
    // Returns:
    // The token that is associated with this instance.
    public CancellationToken CancellationToken { get; set; }

    // Gets or sets the maximum number of concurrent tasks
    // enabled by this System.Threading.Tasks.ParallelOptions
    // instance.
    // Returns:
    // An integer that represents the maximum degree of
    // parallelism.
    // Exceptions:
    // T:System.ArgumentOutOfRangeException:
    // The property is being set to zero or to a value that is
    less than -1.
```

```
public int MaxDegreeOfParallelism { get; set; }

    // Gets or sets the System.Threading.Tasks.TaskScheduler
    associated with this System.Threading.Tasks.ParallelOptions

    // instance. Setting this property to null indicates that
    the current scheduler

    // should be used.

    // Returns:
    // The task scheduler that is associated with this instance.

    public TaskScheduler TaskScheduler { get; set; }

}
```

Let's modify our parallel code written above to have cancellation support and limit the CPU usage. The modified code is shown below:

```
try
{
    var cts = new CancellationTokenSource();
    var parallelOptions = new ParallelOptions()
    {
        MaxDegreeOfParallelism = 4,
        CancellationToken = cts.Token
    };

    Parallel.ForEach(numbers, parallelOptions, number =>
    {
        CheckEvenAndWrite(number);
    });
}

catch (AggregateException ex)
{
    Logger.Log(ex);
}
```

The `Parallel` class handles all the heavy-duty items like partitioning of data behind the scenes using the default partitioner, scheduling of threads on the `ThreadPool`, state management, and so on. These are advanced topics and outside the scope of

this book. However, the exercise would have links to official MSDN documentation for the curious readers to read and enhance their depth of knowledge.

Following are a few of the considerations, before replacing your for and foreach loops with the parallel counterparts:

1. Ensure that it is fine to execute the operations concurrently and out of order.
2. Ensure that any kind of shared resource or shared data is not modified concurrently; else, it may lead to issues. One way to prevent this is by using synchronization constructs, but then only one thread would run the loop at a time, and the benefits of parallelism would be negated.
3. Ensure that the loop iteration is large enough to use parallelism. For a small number of iterations, parallelism is counterproductive.
4. First, write the sequential code, achieve the functionality that you wish to achieve and then try to see if it needs to be parallelized and can be parallelized.

There are several overloads of `For`, `ForEach`, that takes several additional parameters that can be leveraged to stop or break the loop execution, monitor the state of loop, cancellation support, control the degree of parallelism, and so on. A shortlist of helper types that help to enable this functionality are:

- **ParallelLoopState**: Allows different iterations of a parallel loop to interact with each other and exposes methods to stop, break, exit the iteration, and also expose properties to check if the iteration has an exception or is stopped.
- **ParallelLoopResult**: It's a struct that exposes the completion status of the parallel loop. It has two main properties `IsCompleted` and `LowestBreakIteration`, seeing which we know if the loop completed or exited via stop or break methods.
- **ParallelOptions**: Options to support cancellation, specify scheduler, and to control the degree of parallelism in the parallel loops.
- **CancellationToken**: Token to support cancellation in the parallel loops.

With the feature-rich APIs offered by TPL, developers can focus their energies on programming the application rather than worrying about thinking strategies to do parallel programming.

PLINQ also provides a way to do data parallelism, which we will discuss in the section on PLINQ.

Let's see how we can parallelize the tasks using TPL.

Task parallelism

As the name suggests, task parallelism refers to tasks running concurrently. We have already discussed the task at length earlier in this chapter and the previous chapter as well. Task offers more programmatic control, compared to threads and represents an asynchronous operation. There is yet another useful API called `Invoke` on the `static Parallel` class, which can be used to parallelize the tasks.

Suppose I have the following code:

```
Method1();
Method2();
Method3();
Method4();
```

Each of these methods is independent of the other and doesn't depend on any of the other methods or interfere with their work. However, in traditional programming, I need to run it one after the other. So, if methods take, 500ms, 600ms, 1s, 400ms respectively, then the above code would take at the minimum 2.5s to complete. It is the sheer wastage of time and the non-utilization of CPU resources. With the `Invoke` API of `Parallel` class, I can have these methods run in parallel, by the following code:

`Parallel.Invoke`

```
(
    () => Method1(),
    () => Method2(),
    () => Method3(),
    () => Method4()
);
```

Now, if we measure the time taken by code, it would be somewhere between 1s-1.5s, which is way less than 2.5s, and hence a significant improvement in the time taken. The above code is an example of imperative task parallelism. In the above example, I demonstrated the methods being parallelized, but it's equally valid for mutually independent statements as well, which do not modify the shared resources.

The next important component of the Parallel Extensions library is PLINQ, so let's check it out. **LINQ (Language Integrated Query)** has been around since .NET 3.5, so I assume that the reader is already familiar with LINQ. If not, please read the documentation of LINQ from <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/> before proceeding further.

PLINQ

To put it simply:

$$\text{Parallel} + \text{LINQ} = \text{PLINQ}$$

PLINQ is a technology that allows developers to easily leverage multi CPU cores that today's modern hardware offers. The great thing about PLINQ is that if you are using LINQ-to-objects, there is very little code change needed to convert it to PLINQ and leverage its advantages. To use PLINQ, all you need to do is just adding `AsParallel()` (an extension method on `IEnumerable`) to your collection in your LINQ query. It does the magic and turns the query into a PLINQ query and uses the PLINQ execution engine when executed. With just one minor code change, the code can now utilize the hardware optimally. What does the `AsParallel` extension method do?

`AsParallel` is an extension method in the `ParallelEnumerable` class in `System.Linq` namespace and is defined as:

```
// Enables parallelization of a query.  
// Returns the source as a ParallelQuery to bind to ParallelEnumerable  
// extension methods.  
  
public static ParallelQuery AsParallel(this IEnumerable source);  
  
// Enables parallelization of a query.  
// Returns the source as a System.Linq.ParallelQuery to bind to  
// ParallelEnumerable extension methods.  
  
public static ParallelQuery<TSource> AsParallel<TSource>(this  
    IEnumerable<TSource> source);  
  
// Enables parallelization of a query, as sourced by a custom  
// partitioner that is responsible for splitting the input sequence into  
// partitions.  
// Returns the source as a ParallelQuery to bind to ParallelEnumerable  
// extension methods.  
  
public static ParallelQuery<TSource> AsParallel<TSource>(this  
    Partitioner<TSource> source);
```

`AsParallel()` works by returning a `ParallelQuery` and enables parallelization of the query. PLINQ enables declarative data parallelism natural as can be seen in the following sample:

```
IEnumerable<int> numbers = Enumerable.Range(1, 100000);
```

```
IEnumerable<int> evenNumbers = numbers.AsParallel().Where(j => j % 2 == 0);
```

Since we have seen constructs of parallel programming, it is also essential to understand the data structures to be used to avoid unnecessary issues that may come up while updating or writing to a shared resource concurrently.

Data structures for parallelism

The `System.Collections.Concurrent` namespace contains the data structures that should be used in concurrency scenarios. The conventional data structures like `List<T>`, `Stack<T>`, `Queue<T>`, `Dictionary<T>`, and so on exposed in `System.Collections` and `System.Collection.Generic` namespaces are not thread-safe unless explicitly specified. Generally, the `static` methods are made thread-safe, but instance methods are not, but we must read the documentation before making any sort of assumption. If multiple threads try updating any of these data structures, it may result in data corruption, unexpected results, or even exceptions. Since the parallel extensions enable multiple threads to work on operation concurrently, we may well run into cases where multiple threads operate on these data structures. Hence, we may run into unwanted results if we continue using these non-thread-safe data structures. The following are a few of the data structures that are shipped inside the `System.Collections.Concurrent` namespace and are recommended to be used in case of parallelism and concurrency scenarios:

- **ConcurrentBag<T>**: Thread-safe implementation of the bag. It contains an unordered set of items. It allows duplicate items. Following is the class definition with a few important members and comments to understand them:

```
// Represents a thread-safe, unordered collection of objects.
public class ConcurrentBag<T> : IProducerConsumerCollection<T>,
IEnumerable<T>, IEnumerable, ICollection, IReadOnlyCollection<T>
{
    // Constructor
    public ConcurrentBag();

    // Constructor taking IEnumerable as input to initialize
    // itself.
    public ConcurrentBag(IEnumerable<T> collection);

    // Returns the number of elements contained in the System.
    Collections.Concurrent.ConcurrentBag
```

```
    public int Count { get; }

    // Returns true if the System.Collections.Concurrent.
    ConcurrentBag is empty; otherwise, false.
    public bool IsEmpty { get; }

    // Adds an object to the System.Collections.Concurrent.
    ConcurrentBag
    public void Add(T item);

    // Copies the System.Collections.Concurrent.ConcurrentBag
    elements to an existing
    // one-dimensional System.Array, starting at the specified
    array index.
    public void CopyTo(T[] array, int index);

    // Returns an enumerator that iterates through the System.
    Collections.Concurrent.ConcurrentBag
    public IEnumrator<T> GetEnumerator();

    // Returns A new array containing a snapshot of elements
    copied from the System.Collections.Concurrent.ConcurrentBag
    public T[] ToArray();

    // Attempts to return an object from the System.Collections.
    Concurrent.ConcurrentBag without removing it.
    // Returns: true if an object was returned successfully;
    otherwise, false.
    public bool TryPeek(out T result);

    // Attempts to remove and return an object from the System.
    Collections.Concurrent.ConcurrentBag
    // Returns: true if an object was removed successfully;
    otherwise, false.
    public bool TryTake(out T result);
}
```

- **ConcurrentStack<T>**: Thread-safe implementation of a stack. Processes items in last-in, first-out (LIFO) order. Following is the class definition with a few important members and comments to understand them:

```
// Represents a thread-safe last in-first out (LIFO) collection.  
public class ConcurrentStack<T> : IProducerConsumerCollection<T>,  
IEnumerable<T>, IEnumerable, ICollection, IReadOnlyCollection<T>  
{  
    // Constructor.  
    public ConcurrentStack();  
  
    // Constructor taking IEnumerable<T> as parameter to  
    // initialize itself.  
    public ConcurrentStack(IEnumerable<T> collection);  
  
    // Returns the number of elements contained in the  
    // System.Collections.Concurrent.ConcurrentStack.  
    public int Count { get; }  
  
    // Returns true if the System.Collections.Concurrent.  
    // ConcurrentStack is empty; otherwise, false.  
    public bool IsEmpty { get; }  
  
    // Removes all objects from the System.Collections.  
    // Concurrent.ConcurrentStack.  
    public void Clear();  
  
    // Copies the System.Collections.Concurrent.  
    // ConcurrentStack elements to an existing 1-D System.Array,  
    // starting at the specified array index.  
    public void CopyTo(T[] array, int index);  
  
    // Returns an enumerator that iterates through the  
    // System.Collections.Concurrent.ConcurrentStack  
    public IEnumerator<T> GetEnumerator();  
  
    // Inserts an object at the top of the System.
```

```
Collections.Concurrent.ConcurrentStack
    public void Push(T item);

        // Inserts multiple objects at the top of the System.
Collections.Concurrent.ConcurrentStack atomically.
    public void PushRange(T[] items);

        // Returns a new array containing a snapshot of elements
copied from the System.Collections.Concurrent.ConcurrentStack.
    public T[] ToArray();

        // Attempts to return an object from the top of the
System.Collections.Concurrent.ConcurrentStack without removing
it.
    public bool TryPeek(out T result);

        // Attempts to pop and return the object at the top of
the System.Collections.Concurrent.ConcurrentStack.
    public bool TryPop(out T result);

        // Attempts to pop and return multiple objects from
the top of the System.Collections.Concurrent.ConcurrentStack
atomically.
    public int TryPopRange(T[] items);
}
```

- **ConcurrentQueue<T>**: Thread-safe implementation of a queue. Processes items in first-in, first-out (FIFO) order. Following is the class definition with a few important members and comments to understand them:

```
// Represents a thread-safe first in-first out (FIFO) collection.
public class ConcurrentQueue<T> : IProducerConsumerCollection<T>,
IEnumerable<T>, IEnumerable, ICollection, IReadOnlyCollection<T>
{
    // Constructor.
```

```
public ConcurrentQueue();

    // Constructor taking IEnumerable as input to initialize
    // itself.
    public ConcurrentQueue(IEnumerable<T> collection);

    // Gets the number of elements contained in the System.
    // Collections.Concurrent.ConcurrentQueue.
    public int Count { get; }

    // Gets a value that indicates whether the System.
    // Collections.Concurrent.ConcurrentQueue is empty.
    public bool IsEmpty { get; }

    // Copies the System.Collections.Concurrent.ConcurrentQueue
    // elements to an existing one-dimensional System.Array, starting at
    // the specified array index.
    public void CopyTo(T[] array, int index);

    // Adds an object to the end of the System.Collections.
    // Concurrent.ConcurrentQueue.
    public void Enqueue(T item);

    // Returns an enumerator that iterates through the System.
    // Collections.Concurrent.ConcurrentQueue.
    public IEnumerator<T> GetEnumerator();

    // Copies the elements stored in the System.Collections.
    // Concurrent.ConcurrentQueue
    public T[] ToArray();

    // Tries to remove and return the object at the beginning of
    // the concurrent queue.
    // Returns true if an element was removed and returned
    // from the beginning of the System.Collections.Concurrent.
    // ConcurrentQueue successfully; otherwise, false.
    public bool TryDequeue(out T result);

    // Tries to return an object from the beginning of the
```

System.Collections.Concurrent.ConcurrentQueue without removing it.

```
// Returns true if an object was returned successfully;  
otherwise, false.  
  
    public bool TryPeek(out T result);  
}
```

- **ConcurrentDictionary<T>**: Thread safe implementation of dictionary. Contains unordered set of key and value pairs. Following is the class definition with few important members and comments to understand them:

```
// Represents a thread-safe collection of key/value pairs that  
can be accessed by multiple threads concurrently.  
  
public class ConcurrentDictionary<TKey, TValue> :  
IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey,  
TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable,  
IDictionary, ICollection, IReadOnlyDictionary<TKey, TValue>,  
IReadOnlyCollection<KeyValuePair<TKey, TValue>>  
{  
    // Constructor  
    public ConcurrentDictionary();  
  
    // Constructor which takes key value collection as parameter  
    to initialize itself  
    public ConcurrentDictionary(IEnumerable<KeyValuePair<TKey,  
TValue>> collection);  
  
    // Gets or sets the value associated with the specified key.  
    public TValue this[TKey key] { get; set; }  
  
    // Gets the number of key/value pairs contained in the  
    System.Collections.Concurrent.ConcurrentDictionary.  
    public int Count { get; }  
  
    // Gets a value that indicates whether the System.  
    Collections.Concurrent.ConcurrentDictionary is empty.  
    public bool IsEmpty { get; }  
  
    // Gets a collection containing the keys in the System.  
    Collections.Generic.Dictionary.  
    public ICollection<TKey> Keys { get; }
```

```
// Gets a collection that contains the values in the System.  
Collections.Generic.Dictionary.  
public ICollection<TValue> Values { get; }  
  
// Uses the specified functions to add a key/value pair to the  
System.Collections.Concurrent.ConcurrentDictionary  
// if the key does not already exist, or to update  
a key/value pair in the System.Collections.Concurrent.  
ConcurrentDictionary  
// if the key already exists.  
public TValue AddOrUpdate(TKey key, Func<TKey, TValue>  
addValueFactory, Func<TKey, TValue, TValue> updateValueFactory);  
  
// Removes all keys and values from the System.Collections.  
Concurrent.ConcurrentDictionary.  
public void Clear();  
  
// Determines whether the System.Collections.Concurrent.  
ConcurrentDictionary contains the specified key.  
public bool ContainsKey(TKey key);  
  
// Adds a key/value pair to the System.Collections.  
Concurrent.ConcurrentDictionary if the key does not already  
exist.  
public TValue GetOrAdd(TKey key, TValue value);  
  
// Copies the key and value pairs stored in the System.  
Collections.Concurrent.ConcurrentDictionary to a new array.  
public KeyValuePair<TKey, TValue>[] ToArray();  
  
// Attempts to add the specified key and value to the System.  
Collections.Concurrent.ConcurrentDictionary.  
public bool TryAdd(TKey key, TValue value);  
  
// Attempts to get the value associated with the specified key  
from the System.Collections.Concurrent.ConcurrentDictionary.  
public bool TryGetValue(TKey key, out TValue value);  
  
// Attempts to remove and return the value that has
```

the specified key from the `System.Collections.Concurrent.ConcurrentDictionary`.

```
public bool TryRemove(TKey key, out TValue value);
```

```
// Compares the existing value for the specified key with a
// specified value, and if they are equal, updates the key with a
// third value.
```

```
public bool TryUpdate(TKey key, TValue newValue, TValue
comparisonValue);
```

```
}
```

All the preceding data structures are non-blocking in nature; that is, if thread looks up for an element that doesn't exist in the data structure, it doesn't wait (block) for that item to appear in the data structure. The thread returns immediately in such cases. That said, there is no magic inside these collections to make them thread-safe. `ConcurrentDictionary` and `ConcurrentBag` make use of synchronization primitives (discussed in *Chapter 7*) to make them thread-safe, but they acquire the lock for a short duration; hence they will perform better than using synchronization constructs on non-thread-safe counterparts. `ConcurrentStack` and `ConcurrentQueue` are lock-free as they make use of `Interlocked` class for thread safety.

The following table represents the non-thread safe collections and corresponding thread-safe collection exposed by `System.Collections.Concurrent` and are recommended to be used:

Non-thread-safe collection	Thread-safe collection
<code>List<T></code>	<code>ConcurrentBag<T></code>
<code>Dictionary<T,U></code>	<code>ConcurrentDictionary<T,U></code>
<code>Stack<T></code>	<code>ConcurrentStack<T></code>
<code>Queue<T></code>	<code>ConcurrentQueue<T></code>

Table 5.2: Thread-safe collections

Notice that `ConcurrentQueue`, `ConcurrentStack`, and `ConcurrentBag` implements an interface named `IProducerConsumerCollection<T>`, which is designed to handle the classic producer-consumer problem in which multiple threads adds the data (producer), and multiple threads are reading the data (consumer) concurrently. This interface definition is shown below, with comments to make the code easy to understand:

```
// Defines methods to manipulate thread-safe collections intended for
// producer/consumer
// usage. This interface provides a unified representation for producer/
```

```

consumer

// collections so that higher-level abstractions such as System.
Collections.Concurrent.BlockingCollection

// can use the collection as the underlying storage mechanism.

public interface IProducerConsumerCollection<T> : IEnumerable<T>,
IEnumerable, ICollection

{

    // Copies the elements of the IProducerConsumerCollection to a
    System.Array, starting at a specified index.

    void CopyTo(T[] array, int index);

    // Returns a new array containing the elements copied from the
    IProducerConsumerCollection.

    T[] ToArray();

    // Attempts to add an object to the System.Collections.Concurrent.
    IProducerConsumerCollection.

    // Returns true if the object was added successfully; otherwise,
    false.

    bool TryAdd(T item);

    // Attempts to remove and return an object from the
    IProducerConsumerCollection.

    // Returns true if an object was removed and returned successfully;
    otherwise, false.

    bool TryTake(out T item);

}

```

There is a `BlockingCollection<T>` data structure as well in `System.Collections.Concurrent` namespace, which implements `IProducerConsumerCollection<T>` and can, therefore, be used in concurrent producer-consumer scenarios. The collection is called **blocking** because of the thread **blocks** if the item it is looking for is not found and keeps looking for it. The `BlockingCollection<T>` is defined as:

```

// Provides blocking and bounding capabilities for thread-safe
collections that implement IProducerConsumerCollection.

public class BlockingCollection<T> : IEnumerable<T>, IEnumerable,
ICollection, IDisposable, IReadOnlyCollection<T>

```

```
{  
    // Initializes a new instance of the BlockingCollection class  
    // without an upper-bound.  
    public BlockingCollection();  
  
    // Initializes a new instance of the BlockingCollection class with  
    // the specified upper-bound.  
    public BlockingCollection(int boundedCapacity);  
  
    // Initializes a new instance of the BlockingCollection  
    // class without an upper-bound and using the provided  
    // IProducerConsumerCollection as its underlying data store.  
    public BlockingCollection(IProducerConsumerCollection<T>  
        collection);  
  
    // Initializes a new instance of the BlockingCollection  
    // class with the specified upper-bound and using the provided  
    // IProducerConsumerCollection as its underlying data store.  
    public BlockingCollection(IProducerConsumerCollection<T> collection,  
        int boundedCapacity);  
  
    // Gets the bounded capacity of this BlockingCollection instance.  
    public int BoundedCapacity { get; }  
  
    // Gets the number of items contained in the BlockingCollection.  
    public int Count { get; }  
  
    // Gets whether this BlockingCollection has been marked as complete  
    // for adding.  
    public bool IsAddingCompleted { get; }  
  
    //// Other members not shown for brevity  
}
```

As we see from the definition, the class doesn't implement `IProducerConsumerCollection<T>` but takes it as an input parameter in the constructor as a backing store. If the constructor which doesn't take `IProducerConsumerCollection<T>` as parameter is used, then `BlockingCollection<T>` uses `ConcurrentQueue<T>` as `IProducerConsumerCollection<T>` by default.

With this, we conclude our discussion on parallel extensions. In the next section, we will discuss `IEnumerator` and `yield return` as it is the fundamental concept to understand `async await`, which is now the recommended approach for asynchronous programming.

IEnumerator and yield return

Let's have a look at this simple console application code:

```
static void Main(string[] args)
{
    // Get authors
    var authors = GetAuthorNames();

    foreach (var author in authors)
    {
        // Iterate and write their name in Console.
        Console.WriteLine($"{author}");
    }
}

public static IEnumerable<string> GetAuthorNames()
{
    yield return "Rishabh Verma";
    yield return "Neha Shrivastava";
    yield return "Ravindra Akella";
}
```

The code is straightforward. In the `Main()` method, we make a call to a method named `GetAuthorNames()`, which returns an `IEnumerable<string>` containing names. `IEnumerable<string>` has an `IEnumerator<string>`, shown in *Figure 5.1*.

It means we can iterate over it, so we just iterate over the authors and display their names on the console:

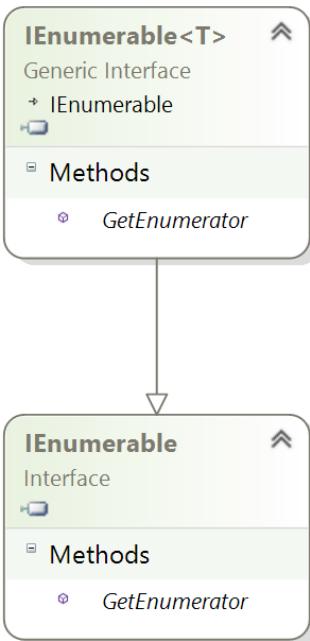


Figure 5.1: Class diagram of `IEnumerable<T>`

The critical thing to notice here is the method `GetAuthorNames()`, which has no collection but returns an `IEnumerable<string>` by using the `yield return` statement. If you put a breakpoint on this method and debug the above program, you will find that method `GetAuthorNames()` is called thrice, and each time it returns different author names. It is the exciting part that even though the method is invoked only once in the code, it is called multiple times. It is the magic of `yield return` statement, which is used to return the elements of a collection one element at a time and prevents the need to maintain a separate collection (`IEnumerable<T>`) to hold the state of iteration. If you can digest this first, understanding `async await` would become more comfortable as `async await` uses a similar mechanism, wherein the execution returns to the caller of the `async` method, as soon as it encounters an `await` statement.

It was a precursor to `async await`, so let's dive right into `async await`.

async await

`async await` is the language features in C# language that helps developers to do asynchronous programming easily. How? Well! C# introduced keywords `async` and

`await` so that developers can write `async` methods directly. Since it's a language feature exposed via keywords, the burden of complicated code is offloaded from developer to the compiler. With `async` and `await` keywords, a developer can just write an `async` method, and the compiler takes the responsibility of writing the complex code behind the scenes and optimizes it as well. Therefore, `async await` is also called syntactic sugar. Let's create our first `async` method using `async await` keywords and understand its working. To do so, first, let us write asynchronous code and then convert it to the `async` method:

```
private static void DownloadData(string url, string path)
{
    // Create a new web client object
    using (WebClient client = new WebClient()) //1
    {
        // Add user-agent header to avoid forbidden errors.
        client.Headers.Add("user-agent", "Mozilla/5.0 (Windows
NT 10.0; WOW64)") //2
        // download data from Url
        byte[] data = client.DownloadData(url); //3
        // Write data in file.
        using (var fileStream = File.OpenWrite(path)) //4
        {
            fileStream.Write(data, 0, data.Length); //5
        }
    }
}
```

We have a simple `DownloadData` method, which accepts two arguments of type `string`, namely `url` and `path`. The code is straightforward and self-explanatory, using comments, but we will still discuss it as it lays the foundation stone for `async await`. Let us see what each line of the above program does. (For the sake of clarity and one to one mapping, each line of the program has line number as the comment):

1. We create a new `WebClient` object. `WebClient`, as the name suggests, is a client that contains the APIs for sending and receiving the data from a web resource. It does so, using sending `HttpRequest` and receiving `HttpRequest`.
2. Next, we set the user-agent header information of `HttpRequest`.
3. The client's `DownloadData` method is invoked, passing `url` as the parameter. Note that this method is synchronous and may run for a while. The thread executing this statement would block and wait until the statement completes

successfully or encounters an error. Eventually, the result would be obtained, and it would be stored in a local variable named `data` of type `byte[]`.

4. We have the data downloaded from the `url`, so we want to save this data in a file path. To do so, we create a new `FileStream` object, passing the path as the parameter.
5. We invoke the `Write` method on the `fileStream` object and write the downloaded data to the file.

Since the `WebClient` and `FileStream` implement `IDisposable`, we have wrapped their object creation code inside the `using` block, which would ensure that once the object is no longer used, the memory of these objects is reclaimed by the CLR via **Garbage Collection (GC)**.

INFO: The `using` block can be used to wrap the types that implement `IDisposable`. It's again a syntactic sugar offered by C#. It translates into try-finally block, to ensure that the `Dispose` method of types implementing `IDisposable` is called, and there is no memory leak. The next two code snippets are roughly the same:

```
// Implementation using "using" block
using (FileStream stream = new FileStream(@"C:\Rishabh\Threading.txt", FileMode.OpenOrCreate))
{
    // Do some work.
}

// Implementation using "try finally" block
FileStream stream = null;
try
{
    stream = new FileStream(@"C:\Rishabh\Threading.txt",
FileMode.OpenOrCreate);
    // Do some work.
}
finally
{
    if (stream != null)
    {
        stream.Dispose();
    }
}
```

C# is a high-level language. The code that we write in C# is for the application layer. For the same reason, it is more or less independent of the type of computer hardware, and anything that we write in C# undergoes multiple transformations before they get converted into assembly language, which the CPU understands and processes. C# doesn't communicate with hardware directly. In the async sample program with method `DownloadDataAsync`, we have two instances where we need to communicate with the computer hardware to do the job. Generally, a programmer need not understand this detail, and only knowing to use the APIs correctly should suffice. However, if we understand these fundamentals, we would be better able to appreciate what `async await` brings to the table.

Inline #3, we call the `DownloadData` method on a web client object, which is synchronous. The `DownloadData` method downloads the data from the specified URL in a byte array. Depending upon the data available in the URL, network speed, and other computer configuration parameters, it may take a while for this method to download the data and return the `byte[]`. Now imagine, if we have an ASP.NET Core or ASP.NET application, in which this method is being executed. The framework would allocate one of the `ThreadPool` threads to run this method for a request. The thread would execute the code till `DownloadData`. In the `DownloadData` method, behind the scenes, the call would go from managed code (C# .NET) to the native code. The native code would talk to the hardware and instruct it to download the data from the specified URL. During this time, the thread allocated by `ThreadPool` in the managed code has nothing to do but just wait and wait. Sometime in the future, the hardware would finish its job and return the data to native code, which would, in turn, return the data in `byte[]` to the managed code. Recall that the operation in which thread's primary task is just to wait for the operation is the I/O bound peration. It indeed is an I/O bound operation. Under the hood, the CLR thread pool makes use of I/O port to schedule the threads for I/O operations. These threads are referred to as **I/O Completion Port (IOCP)** threads. A similar thing happens in line #5 as well. Here the data is written to the hard-disk drive of the computer, which is an I/O operation, and during this time as well, the managed thread is just sitting idle and waiting!

Since the code is running in a web application, it is common to expect a scenario in which bursts of requests arrive at the server. `ThreadPool` would allocate a thread per request, so depending upon the number of requests, we may have a large number of threads trying to execute this code. As we saw above, there would be a period in which all of these threads would just sit and wait for the data to be downloaded and returned to it. Later, the data would be fetched and returned to the thread. Even in this case, depending upon several cores in the server, only that many threads would at best be able to run concurrently, and others would just wait for the context switch to happen. Recall that context switches are expensive and so we have following highly expensive observations in the above code:

1. We are unnecessarily allocating a thread per request. (Threads are expensive).
2. The thread spends a considerable amount of time doing nothing and just waiting for an I/O operation to complete. (Wasting resources).
3. When the data returns, since we have many threads (more than CPU cores), the threads would compete for the context switch to happen to continue further processing (Context switches are expensive).
4. This wasting of resources is happening twice in the method. Line #3 and line #5.

Due to the above reasons, the synchronous I/O operations are not scalable as we may soon reach the memory and CPU limits because we are wasting or not utilizing the resources optimally. It is assuming we have enough threads in the `ThreadPool`; else, there may be other severe issues like thread exhaustion due to `ThreadPool` throttling or `HttpRequest` getting queued. We will discuss these issues in the later chapter on debugging. We will see how leveraging asynchronous methods via `async await` would solve this issue and enable the path for making highly scalable solutions. Before we dive into the `async` version of the above code, let us quickly see how we can invoke the above code from a Console App:

To call this method from the Console App, we need the following lines of code:

```
static void Main(string[] args)
{
    ServicePointManager.SecurityProtocol = SecurityProtocolType.
Tls12;
    // Set the url to a website from which content needs to be
downloaded.
    string url = "https://bpbonline.com/collections/c-sharp";
    // Path where downloaded data needs to be saved.
    string path = "C:\\Rishabh\\download.txt";

    // Ensure that the directory of the file path exists.
    var directory = System.IO.Path.GetDirectoryName(path);
    if (!Directory.Exists(directory))
    {
        Directory.CreateDirectory(directory);
    }

    // Call the method.
```

```

        DownloadData (url, path);
        // Prevent the program from exiting unless you press enter.
        Console.ReadLine();
    }
}

```

Upon executing this code, the data would be downloaded from the specified URL and dumped into the file name `download.txt` in the specified location. This code is simple and executes synchronously.

Let's convert our `DownloadData` method into the asynchronous method, leveraging `async` `await` keywords. The rewritten method would look like:

```

private static async Task DownloadDataAsync(string url, string
path)
{
    // Create a new web client object
    using (WebClient client = new WebClient()) //1
    {
        // Add user-agent header to avoid forbidden errors.
        client.Headers.Add("user-agent", "Mozilla/5.0 (Windows
NT 10.0; WOW64)"); //2
        // download data from Url
        byte[] data = await client.DownloadDataTaskAsync(url); //3
        // Write data in file.
        using (var fileStream = File.OpenWrite(path)) //4
        {
            await fileStream.WriteAsync(data, 0, data.Length);
        }
    }
}

```

Pretty simple, right! The following changes are worth noting in this rewritten asynchronous method:

- Though it may not always be the case, there is no change in the number of lines of code from the synchronous version of the method.
- There is a new access modifier `async` added to the method definition.
- The return type of method has changed from `void` to `Task`. Though keeping return type as `void` would have compiled as well, but it is highly **NOT**

recommended to code that way. We will discuss this later in our discussion on "Principles for using `async await`".

- We have changed the method name from `DownloadData` to `DownloadDataAsync` to indicate that this method is asynchronous. Though there is no hard and fast rule like this, it is recommended to have `Async` suffix in the method names that are asynchronous, just to make it easier for API consumers as well as for maintenance and readability.
- The two essential operations where data was getting downloaded as well as downloaded data were being written to file now have `await` keyword before their invocation, and they make use of an `async` version of methods instead of their synchronous method counterparts.

With just these few changes, our synchronous method has been changed to the asynchronous method. It is the USP of the `async` and `await` keywords that it makes writing asynchronous methods easier than ever. Though there is a lot that happens behind the scenes, that is abstracted away from the developer, and the compiler does all the hard part and makes the life of the developer easy.

Based on the above description, we can devise a simple step by step technique to convert any synchronous method to asynchronous method, and they are:

1. Introduce the `async` keyword in the method definition.
2. Replace the return type of the method according to the following table:

Synchronous method return type	async method return type
<code>T</code>	<code>Task<T></code>
<code>void</code>	<code>Task</code>
<code>void</code>	<code>void</code> (only for top-level event handlers, like a button click)

Table 5.3: Return types for `async` method

3. Add `Async` suffix in the method name to declare to the world that the method is asynchronous.

Inside the method, look for method invocations that have asynchronous versions available (end with `Async` and has `Task`-based return types). If yes, use the `async` version with the `await` keyword.

Okay! We have the `async` version of the method. But how is it better than the synchronous version of the method we saw earlier. Let's see the line by line execution of code to understand this. Like earlier, we have the line number appended as a suffix in each line of code:

1. We create a new `WebClient` object. `WebClient`, as the name suggests, is a client that contains the APIs for sending and receiving the data from a web resource. It does so, using sending `HttpRequest` and receiving `HttpRequest`. (Same as synchronous version).
2. Next, we set the user-agent header information of `HttpRequest`. (Same as synchronous version).
3. The client's `DownloadDataTaskAsync` method is invoked, passing `url` as the parameter. Note that this method is asynchronous and is prefixed with the `await` keyword. It is where the compiler will play a part. To simplify and make it comprehensive, the thread executing this code would return to the caller method upon encountering the `await` keyword. If it's a graphical user interface (GUI) application and method are invoked from the top-level event handler, then the main thread will return to process the message pump, and hence UI will remain responsive. If it's a server-side code and `ThreadPool` thread is executing it, then this thread will return to the caller function and hence remain available for further processing (instead of sitting there and waiting!) In the future when the `DownloadDataTaskAsync` method completes its work and return the data in `byte[]`, `ThreadPool` may allocate the same or different thread to resume the method from the same place and continue with the rest of the code. In this sense, `async await` enables the `ThreadPool` threads to return to the `caller` method (top-level awaits may return the thread to the pool) and enter the method multiple times (as many times `await` appears).

TIP: To grasp the fundamental of `async await`, I would suggest thinking of `await` keyword as the `ContinueWith` construct that we discussed in the last chapter. The compiler transforms the code after the `await` statement inside a `ContinueWith` construct. As soon as `await` keyword is encountered, the executing thread returns to the caller. Upon completion of that statement, a thread executes the code wrapped inside the `ContinueWith` construct. It can be on a different thread or the same thread depending upon `ConfigureAwait(false)` is used or not used, respectively. It happens for all `await` statements.

1. We have the data downloaded from the `url`, so we want to save this data in a file path. To do so, we create a new `FileStream` object, passing the path as the parameter. (Same as synchronous version).
2. We invoke the `WriteAsync` method on the `FileStream` object and await it to write the downloaded data to the file. It will have the same behavior, as we discussed in Step 3.
3. The compiler transforms the method using `async await` into a state machine, where a thread can enter multiple times. This way, threads remain free because they are used optimally. They are freed up and return to the caller upon encountering `await` statement and can be used elsewhere, increasing

scalability and minimizing resource wastage. Therefore, methods using `async await` makes the code more scalable and are highly recommended to be used in the server-side applications as well on the **graphical user interface (GUI)** based applications to keep the UI responsive.

How would I call this an asynchronous version of the method? We can call this method with minor tweaks. The modified main method to call the `async` version of the method is shown in the following code block:

```
static async Task Main(string[] args)
{
    ServicePointManager.SecurityProtocol = SecurityProtocolType.
Tls12;
    // Set the url to a website from which content needs to be
downloaded.
    string url = "https://bpbonline.com/collections/c-sharp";
    // Path where downloaded data needs to be saved.
    string path = "C:\\Rishabh\\download.txt";

    // Ensure that the directory of the file path exists.
    var directory = System.IO.Path.GetDirectoryName(path);
    if (!Directory.Exists(directory))
    {
        Directory.CreateDirectory(directory);
    }

    // Call the asynchronous method.
    await DownloadDataAsync(url, path);
    // Prevent the program from exiting unless you press enter.
    Console.ReadLine();
}
```

The code is well commented for the reader to understand it. The differences from the earlier `Main` method are highlighted and made bold for easy reference. Notice that I have put an `async` modifier in the `Main` method and changed the return type to `Task`. Until C# 7, it was not possible to use the `async` modifier in the `Main` method, so the developers were forced to use `.Result` or `.Wait()` on the asynchronous methods. But Microsoft changed this, and now, we can add an `async` modifier in the `Main` method, which lets us use `await` keyword in the `Main` method body. That is

what we have done. We have awaited the call to the `DownloadDataAsync` method. The rest of the code remains the same. It's this easy!

If you mark a method as `async` and don't use `await` keyword in the method body, the compiler will complain and throw the following warnings, reminding the developer to await the asynchronous method calls:

CS4014: Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'await' operator to the result of the call.

Ok. We have converted the synchronous method, to asynchronous method easily, but synchronous code is so simple to understand. Just by looking at the code, if I see an asynchronous method call, I can confidently say that in most cases, after the method call completes, only then the next statement will execute. But seeing all these `await` statements, I am not too sure how the execution would work in case of `async await`. So, let's see the control flow of `async await` methods

async await – Control flow

Let's have a look at the following code using `async await`. Please pay special attention to the numbered steps:

```

static async Task Main(string[] args)
{
    // Call the asynchronous method.
    await DownloadDataAsync(url, path);
    // Prevent the program from exiting unless you press enter.
    Console.ReadLine();
}

1 reference | Rishabh Verma, 28 minutes ago | 1 author, 1 change
private static async Task DownloadDataAsync(string url, string path)
{
    // Create a new web client object
    using (WebClient client = new WebClient())
    {
        // Add user-agent header to avoid forbidden errors.
        client.Headers.Add("user-agent", "Mozilla/5.0 (Windows NT 10.0; WOW64)");
        // download data from Url
        byte[] data = await client.DownloadDataTaskAsync(url);
        // Write data in file.
        using (var fileStream = File.OpenWrite(path))
        {
            await fileStream.WriteAsync(data, 0, data.Length);
        }
    }
}

```

Figure 5.2: `async await` Control flow

In the preceding figure, the high-level steps of the execution flow are marked. Let's discuss these steps to understand the control flow.

1. We know that our `Main` method is `async` as we added an `async` modifier in the `Main` method definition. As we will see in the next section, when a

method is marked with an `async` modifier, the compiler transforms the method's code into a type that implements a state machine. And the thread would execute till it encounters `await` and then return to the caller. So, our main thread enters the `Main` method (being entry point) starts execution of code and executes all the code until it encounters the `await` statement, which is marked as *Step 2*.

2. The main thread passes the `url` and `path` parameters and invokes the `DownloadDataAsync`, which is again an asynchronous method and is awaited. `DownloadDataAsync` method internally creates a `Task` object and returns it to the `Main` method. At this point, the `await` keyword wires up the `callback` method `ContinueWith` on this returned `Task` object and passed the method that resumes the state machine, and then the main thread returns from the `Main` method.
3. The `DownloadDataAsync` method would run on the `Main` thread until it encounters its first `await` statement. It can be seen by printing the `Thread.CurrentThread.ManagedThreadId` before and after `await` statements in the `Main` as well as `DownloadDataAsync` method. Upon encountering `await` statement `await client.DownloadDataTaskAsync`, the `Task` object would be created and returned to the `DownloadDataAsync` method, and the `await` keyword wires up the `callback` method `ContinueWith` on the `Task` object and the thread returns to `DownloadDataAsync`.
4. Sometime later, the `HttpClient` will complete downloading the data from the `url`, and a `ThreadPool` thread will notify the `Task` object, which would result in the activation of the `callback` method `ContinueWith` and the thread would resume the method from `await` statement. Now, the `DownloadDataTaskAsync` method of `HttpClient` could have completed the task successfully or may have encountered a network error. All these status checks are done by the compiler-generated code behind the scenes, and the method execution continues on the `ThreadPool` thread, which will then create a `FileStream` and call its `WriteAsync` method. Again, the `await` operator calls a `ContinueWith` on the task object returned from the `WriteAsync` method passing in the `callback` method name to resume the method, and the thread returns from the `DownloadDataAsync` method again.
5. After some time, the write operation would complete. A `ThreadPool` thread will notify the completion, and the thread will resume the `DownloadDataAsync` method until its completion and return to the `Main` method. The compiler will generate the code to ensure that the `Main` method knows that the `DownloadDataAsync` method is now complete, and it has no more `await` statements. It is done by marking the status of `Task` returned from the `DownloadDataAsync` method as `Completed`. The thread then waits at the `console.ReadLine()` is waiting for a user to enter to exit the console.

Now that we have a good idea of `async await` let us see how `async await` works.

async await – Under the hood

Before we go into the intricacies of how `async await` works under the hood, let us list out the key points to remember for `async await`. `async await` is the language features, so the compiler does a lot of work behind the scenes, which would have been rather difficult for a developer to do. Apart from the compiler, the framework also enables us to leverage `async await`.

Language features

- **async:**
 - It's a keyword and used as a modifier in the method definition.
 - Using the `async` keyword marks the method or lambda as asynchronous.
 - The `async` keyword enables us to use the `await` keyword in the method body.
 - It instructs the compiler to transform the method into a state machine.
- **await:**
 - It's a keyword and used as operator to `await` `async` methods in the body of `async` methods or lambdas.
 - It yields the control to the caller until the awaited task completes; that is, whenever an `await` statement is encountered, the execution is immediately returned to the caller of the `async` method.
 - Compiler rewrites `await` statements to use continuations.
- **Framework:**
 - Enables the usage of `Task` and `Task<TResult>` to represent an "ongoing operation," which would complete in the future. It can be an asynchronous I/O operation, background work, or any other work.
 - It provides a single object, which a developer can query to get the status of operation, result, and exceptions.
 - It also provides composable callback methods.

Let us use the same program that we discussed earlier to see what happens under the hood. We will take the build output of the program and load it in ILDASM or JustDecompile or any other reverse engineering tool of your choice. The decompiled IL code of `DownloadDataAsync` method is shown in the following code:

```
.method private hidebysig static class [System.Runtime]System.Threading.Tasks.Task DownloadDataAsync (
    string url,
```

```
        string path
    ) cil managed
{
    .custom instance void [System.Runtime]System.Runtime.
CompilerServices.AsyncStateMachineAttribute::ctor(class [System.
Runtime]System.Type) = (
        01 00 2a 41 73 79 6e 63 41 77 61 69 74 2e 50 72
        6f 67 72 61 6d 2b 3c 44 6f 77 6e 6c 6f 61 64 44
        61 74 61 41 73 79 6e 63 3e 64 5f 5f 31 00 00
    )
    .custom instance void [System.Diagnostics.Debug]System.
Diagnostics.DebuggerStepThroughAttribute::ctor() = (
        01 00 00 00
    )
    .locals init (
        [0] class AsyncAwait.Program/'<DownloadDataAsync>d__1' V_0,
        [1] valuetype [System.Threading.Tasks]System.Runtime.
CompilerServices.AsyncTaskMethodBuilder V_1
    )

    IL_0000: newobj instance void AsyncAwait.
Program/'<DownloadDataAsync>d__1'::ctor()
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: ldarg.0
    IL_0008: stfld string AsyncAwait.
Program/'<DownloadDataAsync>d__1'::url
    IL_000d: ldloc.0
    IL_000e: ldarg.1
    IL_000f: stfld string AsyncAwait.
Program/'<DownloadDataAsync>d__1'::path
    IL_0014: ldloc.0
    IL_0015: call valuetype [System.Threading.Tasks]System.Runtime.
CompilerServices.AsyncTaskMethodBuilder [System.Threading.Tasks]System.
Runtime.CompilerServices.AsyncTaskMethodBuilder::Create()
    IL_001a: stfld valuetype [System.Threading.Tasks]System.
```

```

Runtime.CompilerServices.AsyncTaskMethodBuilder AsyncAwait.
Program/'<DownloadDataAsync>d__1'::'>t__builder'
    IL_001f: ldloc.0
    IL_0020: ldc.i4.m1
    IL_0021: stfld int32 AsyncAwait.
Program/'<DownloadDataAsync>d__1'::'>1__state'
    IL_0026: ldloc.0
    IL_0027: ldfld valuetype [System.Threading.Tasks]System.
Runtime.CompilerServices.AsyncTaskMethodBuilder AsyncAwait.
Program/'<DownloadDataAsync>d__1'::'>t__builder'
    IL_002c: stloc.1
    IL_002d: ldloca.s V_1
    IL_002f: ldloca.s V_0
    IL_0031: call instance void [System.Threading.Tasks]System.
Runtime.CompilerServices.AsyncTaskMethodBuilder::Start<class AsyncAwait.
Program/'<DownloadDataAsync>d__1'>(!!0&)
    IL_0036: ldloc.0
    IL_0037: ldflda valuetype [System.Threading.Tasks]System.
Runtime.CompilerServices.AsyncTaskMethodBuilder AsyncAwait.
Program/'<DownloadDataAsync>d__1'::'>t__builder'
    IL_003c: call instance class [System.Runtime]System.Threading.
Tasks.Task [System.Threading.Tasks]System.Runtime.CompilerServices.
AsyncTaskMethodBuilder::get_Task()
    IL_0041: ret
}

```

If we go through the above code, it will confirm what we discussed in the last section.

We see that the compiler applies an attribute, `AsyncStateMachineAttribute`, which is in `System.Runtime.CompilerServices` namespace. This attribute tells that a state machine type needs to be implemented:

1. In the `.locals init` section, which is nothing but a section where method local variables are assigned a default value, we see two local variables, one of type `class` and other a `value type`. The `class` type that is created by the compiler is `AsyncAwait.Program/'<DownloadDataAsync>d__1'` and its variable is `V_0`, while the `value type` struct is `System.Runtime.CompilerServices.AsyncTaskMethodBuilder` and its variable name are `V_1`. The `class` type above implements the state machine that enables the thread to enter the method multiple times. The `value type` represents the builder of the asynchronous task that returns a `Task`.

2. Next, we see that the `Create` method of `AsyncTaskMethodBuilder` is called, which creates an instance of `AsyncTaskMethodBuilder` class.
3. After the instance is created, it calls the `start` method of the builder passing the state machine as `s` parameter by reference. It starts running the builder on the associated state machine.
4. And finally, it gets the task obtained from the above method call and returns it to the caller.

The state machine code is not shown above, but if you would decompile and see the state machine code, it would become clear, how based on state, the control is returned to the caller and how the method resumes from the same place. The above explanation shows the heavy lifting work that the compiler does in the background to make the coding easier for the developers.

Just to put things in perspective, IL is not user friendly, and it may be difficult for a reader to digest it. So, for simplification and easy understanding, following is the rewritten code which would roughly do the same stuff as IL is doing:

```
private static Task DownloadDataAsync(string url, string path)
{
    // Create a new web client object
    using (WebClient client = new WebClient())
    {
        client.Headers.Add("user-agent", "Mozilla/5.0 (Windows NT 10.0; WOW64)");
        // download data from Url
        var task = client.DownloadDataTaskAsync(url);
        return task.ContinueWith((t) =>
        {
            byte[] data = task.Result;
            using (var fileStream = File.OpenWrite(path))
            {
                var task2 = fileStream.WriteAsync(data, 0, data.Length);
                return task2.ContinueWith((q) =>
                {
                    // Any other code to continue. Doesn't exist
                    // in this case.
                    return;
                });
            }
        });
    }
}
```

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

Notice that in the rewritten method, there is no `async` keyword, which is expected as the `async` method is being translated. Then the code until the `await` statement is found the same as the `async` method. Notice that the `await` statement is replaced concerning a task, and the `ContinueWith` is wired up immediately after it with the remainder of the code. The `byte[]` data that was being returned from the task is extracted from the `Result` property of the task. Post this rest of the code is the same as the asynchronous version, till the next `await` statement is encountered. Again, the same logic is applied. I hope this section gives the reader useful insights into how the `async await` method works under the hood.

Since `async await` methods offer the scalability benefits and make asynchronous programming easy, should I not use it everywhere?

Principles for using `async await`

As you make use of `async await`, you would soon realize that if you have a layered application or multiple nested method calls, you must use `async await` from top to bottom. If you do not comply with this rule and use `.Result` or block it via `.Wait()`, you lose the benefit of `async await` as the thread gets blocked on `.Result` or `.Wait()` and hence not available to process any other operation. Here are a few of the unwritten rules to use `async await` effectively:

- Use `async await` all the way that is, from top to bottom. If you have an `async` method, you can check that all its `caller` and `callee` are `async` as well.
 - Avoid creating `async` methods with the `void` return type. Prefer return type as `Task` instead. The only exception to this rule is event handlers, like button click event handler or so on. The `async void` is only for top-level event handlers. It is because `async void` is a fire and forget the call. The caller would not know when the `async void` method has finished, nor the caller would be able to catch exceptions from `async void` methods (As the caller doesn't know when the method finishes, so by the time the `async void` method returns or throws an exception, the `try...catch` block of the caller may already have been executed, so the exception may not get caught in the caller method and gets posted to the main thread directly).
 - Think if the operation in the method is CPU (compute) bound work or I/O bound. You must use `async await` only if its I/O bound work. For compute-bound, consider using TPL and task infrastructure and not `async void`

- If you are developing or working on a class library, let the library methods announce to the world what they are, that is, if they are synchronous, let them be synchronous. Don't use `Task.Run()` on synchronous library methods and convert them as `async` artificially as mixing synchronous and asynchronous code incorrectly, may lead to deadlocks, especially if they are consumed by .NET framework based applications.
- Consider using `ConfigureAwait(false)` in library methods. Recall that in the last chapter, we discussed `SynchronizationContext`. When we await a method, the thread of execution is returned to the caller of the `async` method. When the awaited operation completes, the execution resumes from the same place. By default, this happens on the same thread that called the `async` method. If it's a **Windows Presentation Foundation (WPF)** application or other windows desktop application, where you have a thread affinity in the sense that only the thread that owns the UI can update the UI controls, it's okay. In other cases, we do not bother which thread does the work as long as the work gets done. For all such cases, using `ConfigureAwait(false)` is recommended. Using `ConfigureAwait(false)` also offers a slight performance benefit as the context doesn't need to be maintained and marshaled. The `ConfigureAwait` method is an instance method defined in `Task` and `Task<TResult>` as:

```
// Configures an awainer used to await this Task.  
// continueOnCapturedContext: true to attempt to marshal the  
continuation back to the original context           captured;  
otherwise, false.  
  
// Returns an object used to await this task.  
  
public ConfiguredTaskAwaitable<TResult> ConfigureAwait(bool  
continueOnCapturedContext);
```

.NET Core doesn't have `SynchronizationContext`, but I still highly recommend making use of `ConfigureAwait(false)` in all the library methods if they are `async`.

While applying the principles, you would notice that there are places where either you cannot use `async await` or find limitations in using `async await`. There are few restrictions on `async await` usage. Let's have a look.

Restrictions on `async await`

Following are a few of the noted restrictions on the usage of `async await`:

- **Property:** `async await` can still be used only with methods. If you try to use `async await` with property getter or setter, you will get following error and code compilation will fail:

`CS0106: The modifier 'async' is not valid for this item`

- **out or ref parameters:** We cannot use out or ref parameters in our `async` methods. If we do, we will get the following error, and code will not compile:
`CS1988: Async methods cannot have ref, in or out parameters`
- **The lock statement doesn't work with async await:** In *Chapter 7*, you would see the details of locking. However, the lock is used for thread synchronization to make the section of code thread-safe in a method. If we have an `async` method, we cannot make use of the `lock` keyword. Trying to use it would throw the following error:

```
error CS1996: Cannot await in the body of a lock statement
```

To fix it, we can use other constructs like `SemaphoreSlim`. It would be discussed with example in *Chapter 7*.

In the last chapter, we discussed CPU bound and I/O bound operations in brief. However, since identifying them is one of the guiding principles to use `async await` this topic deserves a small discussion in this chapter.

CPU (compute) bound versus I/O bound work

There are two types of scenarios for which asynchronous writing code is the recommended way to go:

1. **I/O bound work, or input/output bound work:** I/O bound operation can be easily identified by checking if the code would be waiting for a task to complete before proceeding further? If the answer is yes, its I/O bound work. Few examples are calls to the file system, database, or network calls. In such cases, most of the time is spent waiting. These are perfect use cases for `async await`
2. **CPU or computational-bound work:** CPU bound work is the one that performs an expensive CPU operation. An excellent example of CPU bound work is an expensive business algorithm or an extensive calculation. On the desktop apps, this type of work should be offloaded to a background thread to keep the UI responsive while doing that work. A long-running algorithm on the server also can be offloaded to a background thread. However, it would keep the background thread occupied, and under heavy load, it may lead to a decline in scalability. Therefore, it's also essential to measure any code changes we make to see if it helps or doesn't.

No discussion on threading or parallel programming can be complete without a discussion on deadlock. Let us understand what a deadlock is and how we can avoid it while using `async await`

Deadlock

Don't block on tasks. Blocking is the easiest way to deadlocking. Mixing synchronous (blocking) code and asynchronous code is playing with fire. Suppose we have two operations A and B. If A waits on B for completion and B waits on A to complete, we run into a situation where both the operations would keep waiting on each indefinitely, and no work is done. This situation is referred to as "deadlock." The wrong usage of `async await`, can easily lead to deadlocks in .NET full framework, where `SynchronizationContext` exists. However, in .NET Core, we do not have `SynchronizationContext` altogether, so the chances of deadlock reduce in the usage of `async await`. In *Chapter 9* of debugging, we would see a code example of deadlock and how to identify and debug them. However, as a guiding principle, the below table can be referred to ensure we do not run into a deadlock:

Don't use	Consider using
<code>task.Wait()</code>	<code>await task</code>
<code>task.Result</code>	<code>var result = await task</code>
<code>Task.WaitAll(...)</code>	<code>await Task.WhenAll(...)</code>
<code>Task.WaitAny(...)</code>	<code>await Task.WhenAny(...)</code>
<code>Thread.Sleep(2000)</code>	<code>await Task.Delay(2000)</code> <code>ConfigureAwait(false)</code> in library methods.

Table 5.4: Avoid deadlocks

C#7 and C#8 introduced few more constructs, which can be leveraged in the asynchronous programming. They are `ValueTask` (introduced in C# 7, .NET Core 2.0) and asynchronous streams (introduced in C# 8). Let us conclude this chapter with a quick discussion on these two new constructs.

Asynchronous Streams

Tasks + Enumerable = AsyncEnumerable.

C# 8 ships with a new feature called asynchronous streams. As the name suggests, with this feature, we can asynchronously process the data (that is being returned from network, or database, or disk) as it comes. It is made possible by the new `IAsyncEnumerable`, which enables us to await the `IEnumerable`. So, now we can await the `foreach` loop and process one item at a time and return it rather than waiting for the entire chunk of data to be loaded. Using asynchronous streams in your applications can improve the application responsiveness, user experience, and performance to a great extent, without having to write complex code for it. Though we can create an `async` method and return `Task<IEnumerable<T>>`, we cannot

use `yield return` statement for `IEnumerable<T>` inside the `async` method. This limitation is now overcome with `IAsyncEnumerable`. This interface is defined as:

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

Just like `IEnumerable<T>` has a method `GetEnumerator()` which returns `IEnumerator`, `IAsyncEnumerable<T>` has a method `GetAsyncEnumerator()` which returns `IAsyncEnumerator<T>`. It enables the use of `await`.

Let us see it in action.

```
private static async Task PrintAuthorNamesAsync()
{
    // await foreach - Async stream makes it possible.
    await foreach (var item in GetAuthorNamesAsync())
    {
        Console.WriteLine(item);
    }
}

private static async IAsyncEnumerable<string>
GetAuthorNamesAsync()
{
    // This dictionary is just to give a demo.
    // In the normal scenarios, we would just have ids for which
    names would be fetched from the database.

    Dictionary<int, string> authorIdNameMappings = new
    Dictionary<int, string>() { { 1, "Rishabh" }, { 2, "Neha" }, { 3,
    "Ravindra" } };

    foreach (var id in authorIdNameMappings.Keys)
    {
        /// Simulate Getting name from Web API or network by
        inserting delay.

        await Task.Delay(300);
    }
}
```

```
//yield return the detached data  
yield return authorIdNameMappings[id];  
}  
}
```

In the preceding sample, we have two async methods, `PrintAuthorNamesAsync()` and `GetAuthorNamesAsync()`. In the async Main method, we call `PrintAuthorNamesAsync()`, so this is where our flow begins. This method intends to print the author's names in the Console. The very first line of this method illustrates the concept of `async` streams, which enables the use of `await` with `foreach`. The important thing to note here is that the source collection used in the `foreach` loop is an `IAsyncEnumerable<string>`, which is why we can use `await` with a `foreach` loop. It is because source collection is being returned from `GetAuthorNamesAsync()` asynchronous method, which has a simple `foreach` loop. For illustration purposes, I have kept, keys and values in a dictionary and added a delay of 300 ms to simulate the Web API network call. Finally, we use the `yield return` statement, which returns the execution control flow each time and doesn't require us to create additional `IEnumerable<T>` to store the result. If you put the breakpoints on the `yield return` statement and `Console.WriteLine(item)`, you would see that as soon as the `yield return` statement executes, immediately after that control flow to `Console.WriteLine(item)`. This example illustrates how asynchronous streams can be used to asynchronously process the large data source as the items appear in them.

ValueTask

`Task<T>` is class and hence a reference type. Like all reference types, it takes higher memory allocation than a corresponding value type. `Task<T>` as class offers a variety of benefits, like, it can be awaited multiple times by multiple callers concurrently. We can cache it, store it in a list and do task composition using `WhenAll` and `WhenAny`, and so on. However, in most cases, this flexibility may not be needed.

The general use case of the asynchronous method is to invoke a method and `await` it. In most cases, awaiting multiple times is not needed. Moreover, scenarios in which throughput and performance are of utmost importance, creating a lot of `Task` creates an overhead because of being a reference type. Being a reference type, once the `Task` is completed, the **garbage collector (GC)** is tasked to clean up the resources and objects allocated in `Task` creation. If there are a lot of tasks, the work of GC will increase as lots of objects would have got allocated and hence must be reclaimed. It is one area where the performance gains can be made if we replace the reference type with a value type, as GC doesn't have to bother about value types. This led to the introduction of `ValueTask<TResult>` type in .NET Core 2.0. This type is shipped in `System.Threading.Tasks` namespace now, earlier it was shipped in `System`.

`Threading.Tasks.Extensions` namespace. As the name suggests, it's a value type. Hence, it is defined as a struct. The definition of `ValueTask<T>` in the framework is as shown below:

```
// Provides a value type that wraps a Task and a TResult, only one of
// which is used.

[AsyncMethodBuilder(typeof(AsyncValueTaskMethodBuilder<>))]
public readonly struct ValueTask<[NullableAttribute(2)]TResult> :
IEquatable<ValueTask<TResult>>
{
    // Initializes a new instance of the System.Threading.Tasks.
    ValueTask using the supplied task that represents the operation.

    public ValueTask(Task<TResult> task);

    // Initializes a new instance of the System.Threading.Tasks.
    ValueTask using the supplied result of a successful operation.

    public ValueTask(TResult result);

    // Gets a value that indicates whether this object represents a
    successfully completed operation.

    public readonly bool IsCompletedSuccessfully { get; }

    // Gets a value that indicates whether this object represents a
    completed operation.

    public readonly bool IsCompleted { get; }

    // Gets a value that indicates whether this object represents a
    canceled operation.

    public readonly bool IsCanceled { get; }

    // Gets a value that indicates whether this object represents a
    failed operation.

    public readonly bool IsFaulted { get; }

    // Gets the result.

    public readonly TResult Result { get; }

    // Retrieves a Task object that represents this ValueTask.
```

```
public readonly Task<TResult> AsTask();

// Configures an awainer for this value.
[return: NullableAttribute(new[] { 0, 1 })]
public readonly ConfiguredValueTaskAwaitable<TResult>
ConfigureAwait(bool continueOnCapturedContext);

//// Other members not shown for brevity.

}
```

We can see that:

1. `ValueTask<TResult>` is a `struct` and hence a value type. (All `struct` and `enum` are value types as they derive from `System.ValueType`).
2. It has a read-only property named `Result` of type `TResult` and a method `AsTask()`, which returns `Task<TResult>`.
3. Only one of `TResult` or `Task<TResult>` can be used to construct the `ValueTask<TResult>`, that is, it can wrap either `TResult` or `Task<TResult>`.
4. It has an attribute `AsyncMethodBuilder` on top of it so that it can be used as a return type for `async` methods.
5. If the `async` method runs synchronously and finishes successfully, we can create an instance of `ValueTask<TResult>` `struct`, by directly passing in `TResult` and return it. Only if it runs asynchronously or throws an exception, we need to wrap a `Task<TResult>`. It will become clear in the next sample:

```
private static async ValueTask<byte[]>
DownloadByteDataAsync(string url)
{
    if(string.IsNullOrWhiteSpace(url))
    {
        return default;
    }

    // Create a new web client object
    using WebClient client = new WebClient(); // C# 8 feature.
    // Add user-agent header to avoid forbidden errors.
    client.Headers.Add("user-agent", "Mozilla/5.0 (Windows NT
10.0; WOW64)");
```

```
// download data from Url  
return await client.DownloadDataTaskAsync(url);  
}
```

The other sample is the same asynchronous code that we have been discussing for a while now. Just that, it's only the first part of `DownloadDataTaskAsync` method and we have added a check, that if `url` is `null` or whitespace, then we return the default value of `byte[]` (Ideally an exception must be thrown, but this is for demonstration purpose, so please excuse).

For the time being, please assume that the return type of the above method is `Task<byte[]>` (not `ValueTask<byte[]>`), and that value of `url` parameter passed for invocation is `null`. The method would return the default value of `byte[]`, and even though `Task` is not involved, unnecessarily, a `Task` object would be created and returned, which causes performance overheads. Though it may appear to cause negligible overhead in a console app, when the same code gets executed by many requests in a server-side application, this is a significant overhead.

With `ValueTask<byte[]>`, the above problem is mitigated. If the validation fails, the `ValueTask<byte[]>` would just wrap the `byte[]` and return (so no `Task` is created). When the validation passes and the method complete asynchronously, `ValueTask<byte[]>` would wrap a task object and return. Cool! Right?

That said, please be very cautious while using `ValueTask<T>`. You should avoid using it unless you know/test that your method is falling under a hot path. When you use it, ensure that `ValueTask<T>` is not awaited multiple times or concurrently, and the `Result` property should only be called when the operation is completed. There is a non-generic `ValueTask` as well, which is the counterpart of `Task`, but it should not be used.

Summary

In this chapter, we have covered a lot of ground concerning parallel programming. The key takeaway of this chapter is the understanding of Parallel Extensions, TPL, PLINQ, and `async await`. We saw several samples to understand the fundamentals and learned about new constructs that are introduced in C# 8 and .NET Core 3.1.

The reader should have gained useful insights about the common terminologies used in multithreading and understood how `async await` works under the hood. We also learned the scenarios in which `async await` should and should not be used.

In the next chapter, we will learn about thread-safety and synchronization constructs, so that multithreading code doesn't deliver unwarranted and unexpected results.

Exercise

1. Try to find out real-world examples for the terms:
 - a. Parallelism
 - b. Concurrency
 - c. Asynchronous
2. Find out how many cores does your computer has.
3. Write a program to compute the factorial of the first 20 integers and print it on file. Notice the maximum, minimum, and average CPU used by the program. How many cores of the CPU are you utilizing?
4. Write any program to utilize only two cores (in a 4-core machine) of your machine.
5. Write an asynchronous method to read the content of a file and upload it into any other data storage.
 - a. Decompile this program using a reverse engineering tool of your choice and understand the generated transformation. If you find difficulty understanding, read this old, but excellent post on `async await` by Jon Skeet <https://codeblog.jonskeet.uk/category/eduasync/>
6. Read the MSDN documentation for parallel programming to get in-depth coverage of understanding. <https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl>
7. Write a WPF application (in .NET full framework) having a label, button, and its click handler. Write the `async` method (written above) to read the contents of a file and display it in the UI label. On click of a button, this `async` method needs to be called, and file contents need to be displayed in the label. Try the following variations and make a note of results:
 - a. Call `.Wait()` on the `async` method invocation.
 - b. Call `.Result` on the `async` method invocation.
 - c. Make the button click event handler `async` and `await` the `async` method.
 - d. Make the button click event handler `async` and `await` the `async` method and use `ConfigureAwait(false)`.
 - e. In what cases can deadlock occur?
8. Where would you use `async` streams, and why?
9. Where would you use `ValueTask<T>`, and why?
10. Inculcate the habit of reading the .NET blogs regularly. Here is the link <https://devblogs.microsoft.com/dotnet/>.

CHAPTER 6

The Threading Patterns

"To understand is to perceive patterns"

—Anonymous

Introduction

Patterns are tried, tested, and recommended way of implementing a solution for a standard problem; till now, we have seen how easy it is to implement an asynchronous method using `async-await`. In this chapter, we will further see in detail on the patterns that are available using `async await` and `tasks` which can be used in implementing enterprise application. We will further investigate exception handling, cancellation, progress tracking in `tasks`. Further, we will touch base on patterns to implement asynchronous methods without `async-await` and how to implement `async-await` wrapper methods on legacy methods.

Structure

- Objectives
- Task-based Asynchronous Pattern (TAP)
 - Overview
 - Implement TAP

- o CPU Bound versus I/O Bound
- o Exception handling
- o Cancellation
- o Progress reporting
- Asynchronous Programming Model (APM)
 - o APM to TAP wrapper
 - o TAP to APM wrapper
- Event-based Asynchronous Pattern (EAP)
 - o EAP to TAP wrapper
- Summary
- Exercise

Objectives

By the end of this chapter reader should be able to understand:

- What are the various threading patterns available in .NET to develop asynchronous applications?
- How to implement each of these patterns in modern application development?
- How to implement wrapper over legacy patterns if the need arises?

Task-based Asynchronous Pattern (TAP)

Task-based Asynchronous Pattern, also known as **TAP**, is the modern way of implementing asynchronous methods in enterprise applications. TAP pattern is dependent on the instance of `Task`, `Task<T>` types or any type that exposes a `GetAwaiter()` method and represents an asynchronous operation through the instance of the `Task`. This pattern primarily recommends creating a single method for asynchronous operation and return it as a `Task`.

Implementing pattern

To implement this pattern, we will start with prefixing function with `async` keyword and add `await` keyword to the method that can be performed asynchronously - typically a method retrieving data from the database, reading the file from disk or an API call (I/O Bound). It is illustrated in the following example where we have a button click event on a win form loading data from API synchronously:

```
private void Search_Click(object sender, EventArgs e)
{
```

```
BindingSource bindingSource1 = new BindingSource();
var ticker = new Stopwatch();
ticker.Start();
var request = WebRequest.Create("https://localhost:44394/api/
StockSynchronous");
var response = request.GetResponse();
Stream dataStream = response.GetResponseStream();
StreamReader reader = new StreamReader(dataStream);
string responseFromServer = reader.ReadToEnd();
var data = JsonConvert.
DeserializeObject<IEnumerable<Stock>>(responseFromServer);
bindingSource1.DataSource = data.Where(price => price.StockName ==
searchText.Text);
stockData.AutoResizeColumns(DataGridViewAutoSizeColumnsMode.
AllCellsExceptHeader);
stockData.DataSource = bindingSource1;
progressMessage.Text = $"Loaded stocks for {searchText.Text} in
{ticker.ElapsedMilliseconds}ms";
}
```

To convert the above method to an asynchronous method, we prefix it with `async` and also add asynchronous methods with `await` for all the outbound calls. The button click event on a win form loading data from API asynchronously will look like below:

```
private async void Search_Click(object sender, EventArgs e)
{
    BindingSource bindingSource1 = new BindingSource();
    var ticker = new Stopwatch();
    ticker.Start();

    using (HttpClient client = new HttpClient())
    {
        var response = await client.GetAsync($"https://localhost:44394/api/
StockS");
        var content = await response.Content.ReadAsStringAsync();
        var data = JsonConvert.DeserializeObject<IEnumerable<Stock>>(content);
```

```
bindingSource1.DataSource = data.Where(price => price.StockName ==  
searchText.Text);  
}  
stockData.DataSource = bindingSource1;  
progressMessage.Text = $"Loaded stocks for {searchText.Text} in {  
    ticker.ElapsedMilliseconds}ms";  
}
```

In the above sample code, `await` keyword helps to get the result from asynchronous operation once data is available without blocking the UI thread. So, `await` keyword stores result of the `async` operation in the left-hand-side variable as, in this case, the content variable is a string. The benefit of doing this is that UI thread is returned to the caller and unblocks the UI while data is retrieved from API.

Note: `async void` is allowed only for UI event handlers; other scenarios should be avoided; we will see on why to avoid later in this chapter.

CPU bound versus I/O bound

When implementing asynchronous code especially on the server side it is essential to identify whether the method is doing I/O bound task or CPU bound task, a simple way to do is to ask whether my method completion is dependent on external sources, for example, a database call, an API call or load data from a file on disk, `async` is the best fit in such scenarios. However, if you are doing an expensive computational work like executing a business algorithm, `async` is not the best fit as the code will still run synchronously. Let's see that with an example:

Let us create a console application and name it `CPUBoundvsIOBound`; this console application will look like below:

```
using System;  
using System.Net.Http;  
using System.Threading;  
using System.Threading.Tasks;  
  
namespace CPUBoundvsIOBound  
{  
    class Program  
    {  
        static void Main()
```

```
{  
}  
}  
}  
}
```

Now we will add two methods to this class.

GetStocksAsync(): This is a simple async method that makes a call at to an API asynchronously and prints message on success/failure. This method will look like:

```
/// <summary>  
/// Async method to retrieve data from API  
/// </summary>  
static async Task GetStocksAsync()  
{  
    using (HttpClient client = new HttpClient())  
    {  
        try  
        {  
            var response = await client.GetAsync("https://  
localhost:44394/api/Stocks");  
            response.EnsureSuccessStatusCode();  
            var content = await response.Content.  
ReadAsStringAsync();  
            Console.WriteLine("Data retrieved from API");  
        }  
        catch (Exception ex)  
        {  
            Console.WriteLine($"exception occurred in API - {ex.  
Message}");  
        }  
    }  
}
```

Add another method **DoExpensiveCalculationAsync()**, this method will do some in memory calculation asynchronously and consumes significant CPU through a for loop. Code for this method will look like below:

```
/// <summary>
/// Method performing high CPU intense calculation
/// </summary>
static async Task<double> DoExpensiveCalculationAsync()
{
    Console.WriteLine("Start CPU Bound asynchronous task");
    float calculation = 0;
    var output = await Task.Run(() =>
    {
        for (int i = 0; i < 100; i++)
        {
            calculation = calculation * 20;
        }
        return calculation;
    });
    Console.WriteLine("Finished CPU bound Task");
    return output;
}
```

Now add another helper method that calculates number of available I/O and CPU threads using `System.Threading.ThreadPool` class. Definition of this method will look like below:

```
/// <summary>
/// Method to log available threads
/// </summary>
static void AvailableThreads()
{
    int worker, io;
    ThreadPool.GetAvailableThreads(out worker, out io);

    Console.WriteLine("Thread pool threads available at startup: ");
    Console.WriteLine("    Worker threads: {0:N0}", worker);
    Console.WriteLine("    Asynchronous I/O threads: {0:N0}", io);
}
```

With this, update the primary method where we will call both the methods `GetStocksAsync()` and `DoExpensiveCalculationAsync()` and print the number of available threads using `AvailableThreads()`. We will use `await` here for the asynchronous calls; hence signature of the `Main` method will change to `async Task Main()`. With this updated code for the `Main` method will look like below:

```
static async Task Main()
{
    Console.WriteLine("Before I/O bound task");
    Console.WriteLine("=====");
    AvailableThreads(); //Check number of threads initially
    await GetStocksAsync(); //Call async method
    Console.WriteLine("After I/O bound task");
    Console.WriteLine("=====");
    AvailableThreads(); //Check number of threads
    await DoExpensiveCalculationAsync(); //Call a method that
    does CPU intense operation
    Console.WriteLine("After CPU bound task");
    Console.WriteLine("=====");
    AvailableThreads(); //Check number of threads
    Console.ReadLine();
}
```

Running the above code will give output, as shown in *Figure 6.1*:

```
Before I/O bound task
=====
Thread pool threads available at startup:
  Worker threads: 32,767
  Asynchronous I/O threads: 1,000
Data retrieved from API
After I/O bound task
=====
Thread pool threads available at startup:
  Worker threads: 32,767
  Asynchronous I/O threads: 999
Start CPU Bound asynchronous task
Finished CPU bound Task
After CPU bound task
=====
Thread pool threads available at startup:
  Worker threads: 32,766
  Asynchronous I/O threads: 1,000
```

Figure 6.1: Output of threads used for CPU bound versus I/O bound tasks

Here you can see that although we are calling a background thread for CPU intensive operation, it has used a worker thread which is ok for client-side application for things like unblocking UI. However, for an ASP.NET application, this will not provide any significant gain and may also lead to possible deadlock, because this is no different than running the operation synchronously. There is no added benefit of assigning a dedicated thread for CPU bound operation.

So to conclude on I/O versus CPU bound tasks let us take an analogy of buying tickets at movie counter (assuming this is the only way to book tickets):

- You can tell your friend to buy popcorn while you are waiting in a queue.
- However, to buy tickets, there is no alternative but to reach the counter, even assuming multiple counter scenarios (multiple cores) where several people ahead of you are the same across counters; switching across counters is not going to save any additional time.

Understanding CPU bound versus I/O bound task is a crucial deciding factor on how we can gain the benefit of asynchronous programming, as making CPU bound method asynchronous is not going to boost the performance of the application significantly whereas making an I/O call is going to give us a significant overall efficiency in the application.

Note: In reality, there is no thread dedicated for I/O operations because we do not need dedicated CPU time, as time spent is primarily receiving data over the network or reading data from disk so your application will use available resources more effectively leading to better responsiveness faster loading time, and so on.

Exception handling

Exception handling in async methods based on TAP pattern is nothing different than exception handling in any other method in C#, that is, add a `try...catch...finally` block to your code, and you are good to go. Although this is an oversimplification of exception handling, this is the benefit of using `async` and `await` keywords for your asynchronous operations, here compiler is taking care of chaining exception back to the caller and unwrapped exception is thrown back to the calling method. Let us see this with a simple example, as shown below, where we create an asynchronous method calling an API and is throwing an exception and will have a caller method where we are doing basic exception handling. Let us create a console application and add method `GetDataAsync` that retrieves data from an API as shown below:

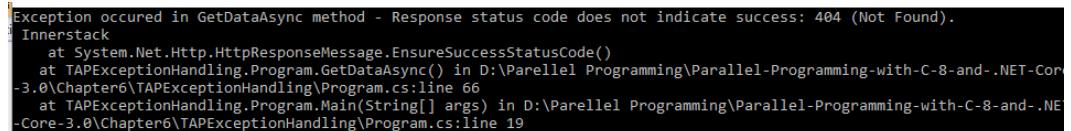
```
using System.Collections.Generic;
using System.Net.Http;
using System.Threading.Tasks;
```

```
namespace TAPExceptionHandling
{
    class Program
    {
        static async Task Main(string[] args)
        {
            var task = GetDataAsync();
            try
            {
                var data = await task;
                Console.WriteLine(data);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Exception occured in GetDataAsync
method - {ex.Message} \n Innerstack \n {ex.StackTrace}");
            }
            Console.Read();
        }

        /// <summary>
        /// Async method to retrieve data from API
        /// </summary>
        /// <returns></returns>
        static async Task<string> GetDataAsync()
        {
            using (HttpClient client = new HttpClient())
            {
                try
                {
                    var response = await client.GetAsync("https://
localhost:44394/api"); // Giving a non-existing API method to generate
exception
                    response.EnsureSuccessStatusCode();
                }
            }
        }
    }
}
```

```
        var content = await response.Content.  
ReadAsStringAsync();  
        Console.WriteLine($"Data retrieved from API");  
        return content;  
    }  
    catch  
    {  
        throw;  
    }  
}  
}  
}
```

Once we run the above code, we will see the output, as shown in *Figure 6.2*:



```
Exception occurred in GetDataAsync method - Response status code does not indicate success: 404 (Not Found).  
Innerstack  
    at System.Net.Http.HttpResponseMessage.EnsureSuccessStatusCode()  
    at TAPExceptionHandling.Program.GetDataAsync() in D:\Parellel Programming\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter6\TAPExceptionHandling\Program.cs:line 66  
    at TAPExceptionHandling.Program.Main(String[] args) in D:\Parellel Programming\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter6\TAPExceptionHandling\Program.cs:line 19
```

Figure 6.2: Basic exception handling

Another way to retrieve exception is to read the exception property of `Task` variable, in above example it's `task.Exception`. So, let us add the following code to the `catch` block of `Main` method:

```
List<String> errors = task.Exception.Flatten().InnerExceptions.Select(x  
=> x.Message).ToList();  
  
int counter = 0;  
  
foreach (string error in errors)  
{  
    counter++;  
    Console.WriteLine($"{counter}).Error - {error}");  
}
```

Once we run this code, we will see the same output as shown in *Figure 6.2*, that is, the inner stack of the exception getting printed just like with the previous code. So, in this sample, we have seen a simple way to handle exceptions in asynchronous code in the next sections, we will see how to handle exceptions in more complex scenarios.

Note: Calling `async` method without `await` is not going to propagate exception and will be swallowed.

Nested exception handling

In the real scenario, there would be more than one asynchronous calls / multiple tasks, handling that wouldn't be any different, that is, use `await`, and the original exception is unwrapped and is propagated to the caller. It is illustrated in the below example where we create a simple console application; let us call it `TAPExceptionHandling`. This application will have three methods out of which the first method will be a method getting data from API asynchronously, adding that to this console application code will look like below:

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

namespace TAPExceptionHandling
{
    class Program
    {
        static void Main(string[] args)
        {

        }

        /// <summary>
        /// Async method to retrieve data from API
        /// </summary>
        /// <returns></returns>
        static async Task<string> GetDataAsync()
        {
            using (HttpClient client = new HttpClient())
            {
                try
```

```
    {
        var response = await client.GetAsync("https://localhost:44394/api"); // Giving a non existing API method to generate exception
        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();
        Console.WriteLine($"Data retrieved from API");
        return content;
    }
    catch
    {
        throw;
    }
}
}
}
}
```

Now add another helper method that will just call `GetDataSync()`, let's call this `GetDataSyncNested()`. The purpose of this call is to simulate hierarchy in function call, that is, simulate nesting:

```
/// <summary>
/// Dummy nested method
/// </summary>
/// <returns></returns>
static async Task<String> GetDataAsyncNested()
{
    return await GetDataAsync();
}
```

Now create another method which retrieves data from a file in this case a non-existing file so that it can throw an exception, this method will look like the following code:

```
/// <summary>
/// Async method to retrieve data from API
/// </summary>
```

```
/// <returns></returns>
static async Task<string> GetDataAsyncFromAnotherSource()
{
    try
    {
        using (var stream = new StreamReader(File.
OpenRead(@"nonexistingfile.txt")))
        {
            var fileText = await stream.ReadToEndAsync();
            Console.WriteLine("Reading from file completed");
            return fileText;
        }
    }
    catch
    {
        throw;
    }
}
```

With all this, let us update the `Main` method where we call both these methods, that is, nested one calling API and another method calling non-existing file. It will simulate a scenario of nested and multiple exceptions, and the way we need to handle it is by adding a `try...catch` block across the tasks and in the `catch` loop through to write it to the console. With that `Main` method will look like below:

```
static async Task Main(string[] args)
{
    var taskfromAPI = GetDataAsyncNested();
    var taskFromFile = GetDataAsyncFromAnotherSource();
    var tasks = new List<Task<string>>();
    tasks.Add(taskfromAPI);
    tasks.Add(taskFromFile);
    var allTasks = Task.WhenAll(tasks);
    try
    {
        await allTasks;
```

```
        }

        catch
        {

            List<Tuple<string, string>> errors = allTasks.Exception.
Flatten().InnerExceptions.Select(x => new Tuple<string, string>(x.
Message, x.StackTrace)).ToList();

            int counter = 0;

            foreach (Tuple<string, string> error in errors)
            {

                counter++;

                Console.WriteLine($"{counter}).Error - {error.Item1}
\n Innerstack \n {error.Item2} \n");

            }
        }

        Console.Read();
    }
}
```

Once we run this code output will look like as shown in *Figure 6.3* where we can see multiple exceptions from asynchronous methods handled and printed on the console (of course in enterprise application, we will do more than printing it to console):

```
1).Error - Response status code does not indicate success: 404 (Not Found).
Innerstack
    at System.Net.Http.HttpResponseMessage.EnsureSuccessStatusCode()
    at TAPExceptionHandling.Program.GetDataAsync() in D:\Parellel Programming\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter6\TAPExceptionHandling\Program.cs:line 93
    at TAPExceptionHandling.Program.GetDataAsyncNested() in D:\Parellel Programming\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter6\TAPExceptionHandling\Program.cs:line 111
    at TAPExceptionHandling.Program.Main(String[] args) in D:\Parellel Programming\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter6\TAPExceptionHandling\Program.cs:line 46

2).Error - Could not find file 'D:\Parellel Programming\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter6\TAPExceptionHandling\bin\Debug\netcoreapp3.0\ nonexistentfile.txt'.
Innerstack
    at System.IO.FileStream.ValidateFileHandle(SafeFileHandle fileHandle)
    at System.IO.FileStream.CreateFileOpenHandle(FileMode mode, FileShare share, FileMode options)
    at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share, Int32 bufferSize, FileMode options)
    at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share)
    at System.IO.File.OpenRead(String path)
    at TAPExceptionHandling.Program.GetDataAsyncFromAnotherSource() in D:\Parellel Programming\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter6\TAPExceptionHandling\Program.cs:line 122
```

Figure 6.3: Nested exception handling

So, this way, we can easily handle multiple or nested exceptions by merely wrapping the asynchronous calls in the `try...catch` block and catching the exception.

Exception handling in Task.Wait()

However, things are different when `await` keyword is not used for an `async` operation and is implemented using `task.Wait()`. All the exceptions are wrapped

in `AggregateException` and thrown to the calling thread. The calling method can specifically catch `AggregateException` to loop through and act accordingly. Let us change the `Main` method in the previous example a bit as shown below, here instead of `await`; we are going to use the `Wait()` method of `Task` for all the task completion and because of which now exceptions are also available in `AggregateException`. With this, our `Main` method will look like below:

```
static void Main(string[] args)
{
    var taskfromAPI = GetDataAsyncNested();
    var taskFromFile = GetDataAsyncFromAnotherSource();
    var tasks = new List<Task<string>>();
    tasks.Add(taskfromAPI);
    tasks.Add(taskFromFile);
    var allTasks = Task.WhenAll(tasks);
    try
    {
        //await allTasks;
        Task.WhenAll(allTasks).Wait();
    }
    catch (AggregateException agex)
    {
        List<Tuple<string, string>> errors = agex.Flatten().InnerExceptions.Select(x => new Tuple<string, string>(x.Message, x.StackTrace)).ToList();
        int counter = 0;
        foreach (Tuple<string, string> error in errors)
        {
            counter++;
            Console.WriteLine($"{counter}).Error - {error.Item1}\n Innerstack \n {error.Item2} \n");
        }
    }
}
```

Once we run this code output of this code is the same as the one in *Figure 6.3*. So, this way, we can additionally catch `AggregateException` and handle it accordingly.

Using the handle method

There could be scenarios where we do not want to propagate specific types of exceptions to parents and do some action in the child method itself. In such cases `AggregateException` gives handle method to filter exceptions and act accordingly, input to it is a function delegate which will be called for each exception, and it needs to be handled within the async method we need to return `true`, else return `false`. In short, the handle method is for "Handled" exceptions, and all unhandled exceptions can be propagated to the calling method, the following example illustrates that behavior. We will use the same console application in the previous example and additionally add another asynchronous method where we handle the specific exception and pass the custom message in case of that exception; this method will loop through the directory and throw an exception which will be handled and thrown to the caller with a custom message. Let's call this method `DoHighCPUIntense()` and definition of this method will look like below:

```
private static string DoHighCPUIntense()
{
    String location = @"C:\";
    Task<string> output = Task.Run(() =>
    {
        List<string> files = new List<string>();
        for (int i = 0; i < 5; i++)
        {
            files.AddRange(Directory.GetFiles(location,
                "*.txt", SearchOption.AllDirectories).ToList());
        }
        return files.FirstOrDefault();
    });
    try
    {
        output.Wait();
    }
    catch (AggregateException agEx)
    {
        //Further handle method can be used to do specific action
        based on the type of exception
    }
}
```

```
        agEx.Handle(x =>
    {
        if (x is UnauthorizedAccessException)
        {
            Console.WriteLine("Specific action for
UnauthorizedAccessException");
        }
        return true;
    });
}
return string.Empty;
}
```

Now modify the `Main` method to call this new method along with the `GetDataAsync` method and catch `AggregateException`; however, this time output will have the custom message. The `Main` method will look like below:

```
static void Main(string[] args)
{
    var tasks = new List<Task>();
    var task = GetDataAsync();
    tasks.Add(task);
    var task2 = Task.Run(() => DoHighCPUIntense());
    tasks.Add(task2);
    try
    {
        Task.WhenAll(tasks).Wait();
    }
    catch (AggregateException agEx)
    {
        List<Tuple<string, string>> errors = agEx.Flatten().
        InnerExceptions.Select(x => new Tuple<string, string>(x.Message,
        x.StackTrace)).ToList();
        int counter = 0;
        foreach (Tuple<string, string> error in errors)
        {
```

```
        counter++;
        Console.WriteLine($"{counter}).Error - {error.Item1}
\n Innerstack \n {error.Item2} \n");
    }
}

Console.Read();
}
```

Here is output in *Figure 6.4* for this method and we can see child method hasn't propagated UnauthorizedAccessException like earlier as it is "Handled" now:

```
Specific action for UnauthorizedAccessException
1).Error - Response status code does not indicate success: 404 (Not Found).
Innerstack
    at System.Net.Http.HttpResponseMessage.EnsureSuccessStatusCode()
    at TAPEXceptionHandling.Program.GetDataAsync() in D:\Parallel Programming\Parallel-Programming-with-C-8-and-.NET-Core
-3.0\Chapter6\TAPEXceptionHandling\Program.cs:line 131
```

Figure 6.4: Exception handling using the handle method

This way further controls the exception that is propagated to the caller method, and this is very useful if we are building libraries or APIs consumed by third-party developers as it helps troubleshoot/debug much quickly from the caller side.

Avoid **async void**

If you have noticed in all examples we discussed until now, some methods aren't returning any data (not even a success flag); however, we still have method signature as an **async Task**. The reason behind that is, one of the advantages of returning **Task** is that entire **async** operation is represented by **Task** object, which in turn also can be used to identify success or failure of the **async** operation. Further to it, any failure can be drilled down based on the exceptions that are placed on the task object and can be handled accordingly. For **async Void** methods as there is no **Task** object, any exception raised would be raised by generic exception handler and can lead to irregular behavior like application crash or w3wp crash in case of the web application.

It is true even in case of fire. It forgets kind of operations like saving data to a central logging data store which is not very critical for user flows and can be **async** operation still it is recommended to avoid using **async Void** as method signature, instead use **async Task** for better maintainability of the application and avoid unexpected crashes in the application. Let's create a simple WinForms application, add a form

and six controls (four text boxes and two buttons). The form should look as shown in *Figure 6.5*:

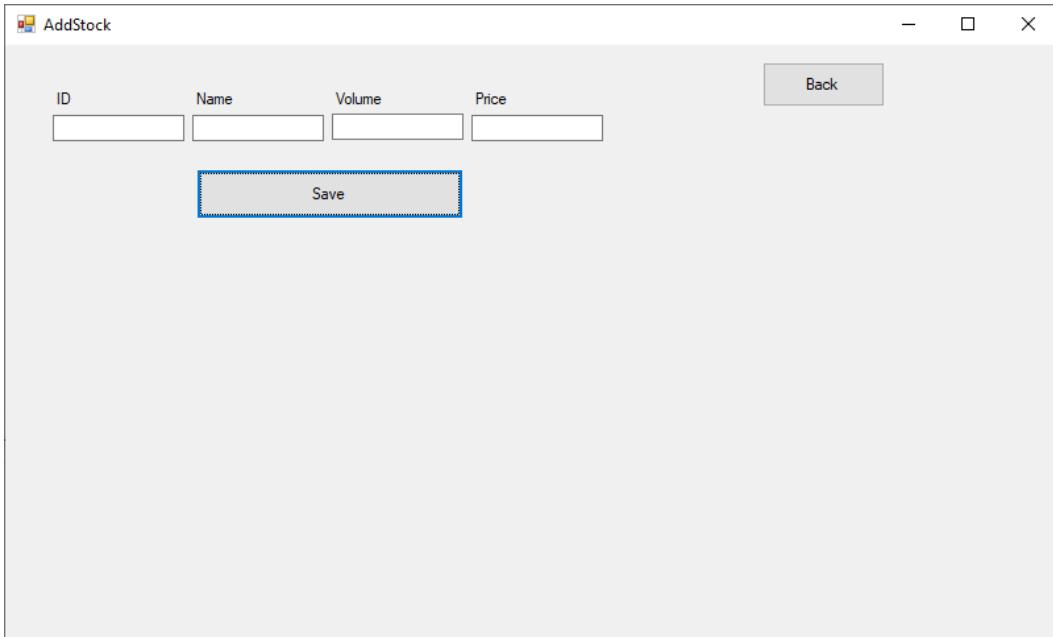


Figure 6.5: Windows form to save data through an API

Add a private helper method to this form that sends data to an API and returns a message of successful save or failure. Let's have the return type of this method as `async void`; this helper method will look like below:

```
/// <summary>
/// Async method to save data through API
/// </summary>
private async void SaveDataAsyncVoid()
{
    using (HttpClient client = new HttpClient())
    {
        Stock data = new Stock()
        {
            Id = Convert.ToInt32(setID.Text),
            StockName = setStockName.Text,
            Price = Convert.ToDouble(setStockPrice.Text),
        }
        // API call logic here
    }
}
```

```
        TradeDate = DateTime.Today.Date.AddDays(-1),
        Volume = Convert.ToInt32(setStockVolume.Text)

    };

    var myContent = JsonConvert.SerializeObject(data);
    var buffer = System.Text.Encoding.UTF8.
GetBytes(myContent);

    var byteContent = new ByteArrayContent(buffer);
    byteContent.Headers.ContentType = new
MediaTypeHeaderValue("application/json");

    var response = await client.PostAsync("https://
localhost:44394/api/stocks", byteContent);

    if (response.StatusCode == HttpStatusCode.
InternalServerError)
    {
        string error = await response.Content.
ReadAsStringAsync();
        throw new Exception(error);
    }

    errorMessage.BeginInvoke((MethodInvoker)delegate () {
        errorMessage.Text = "Data saved successfully";
    });
}
```

Now add a click event handler for the **Save** button, and in this event of **Save** button, we need to make a fire and forget API call to save data using the private helper method created before. Following code shows that:

```
/// <summary>
/// Button click event
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void SaveStock_Click(object sender, EventArgs e)
{
    bool exceptionOccured = false;
```

```
errorMessage.Text = "";

try
{
    SaveDataAsyncVoid();
}

catch (Exception ex)
{
    exceptionOccured = true;

    errorMessage.Text = $"Exception occurred in
SaveDataAsyncVoid method - {ex.Message} \n Innerstack \n {ex.
StackTrace}";

}

finally
{
    setID.Text = "";
    setStockName.Text = "";
    setStockPrice.Text = "";
    setStockVolume.Text = "";
}

if (!exceptionOccured)
    errorMessage.Text = $"SaveStock_Click completed";
}
```

Once you click save if there is an exception in method `SaveDataAsyncVoid()`, you can see that app crashes, and that's because an exception is raised on the `SynchronizationContext` of the UI thread and which won't be present since UI thread has finished its job. Since the method return type is `async void`, we cannot `await` on it either. So to handle this, either we need to build custom `SynchronizationContext` or a better way is to change the method signature to private `async void`, `SaveDataAsyncVoid()` and add `await` in the button click while calling `SaveDataAsyncVoid` and its signature to private `async void SaveStock_Click(object sender, EventArgs e)`. This way, by the time exception is thrown, we still have UI thread with `SynchronizationContext` and get propagated to the caller.

The only exception to having `async void` would be event handlers as event handlers are never called explicitly; that is, the caller of the event handler is not directly interested in the response.

Cancellation

One of the advantages with TAP is the ease with which asynchronous operation can be canceled. Canceling an operation plays a significant role in enhancing user experience. It gives flexibility to users on canceling, for example, consider a search operation in a form with a typo, think about the experience when a user needs to wait for the operation to complete. However, the user is aware that the search result is going to be incorrect because of the typo.

So, when implementing any async method using TAP additionally, a cancellation token can be passed that can be used to cancel the async operation and return to the calling method. Cancellation operation throws an exception of type `OperationCanceledException`, so the calling method needs to handle canceled `async` operation gracefully. It not only gives a better user experience but also frees any thread occupied by I/O bound operation; in case of client-side, it frees any CPU resources used.

The following code illustrates this where we reuse a WinForms application that we created in the previous example and add another form to search stocks from our API; however, cancellation is needed while the search isn't completed. Let us use default form (`Form1`) and add controls to it so that there is a text box to enter stocks that needs to be searched, a button to search stocks, a grid to display stocks, and a text area that displays error messages if any. This form will look like, as shown in *Figure 6.6*:

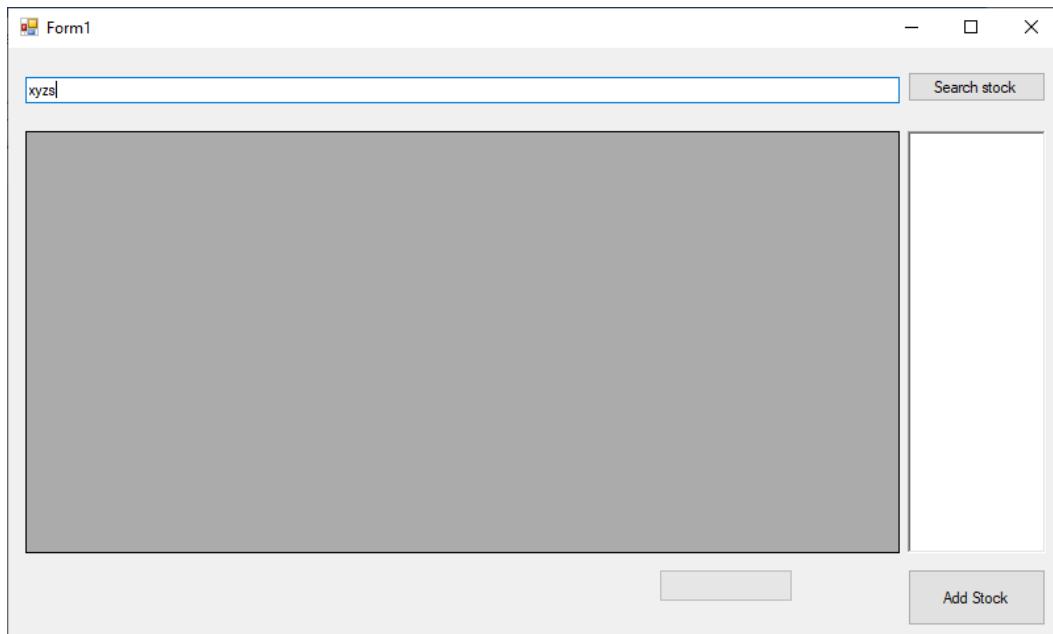


Figure 6.6: Windows form to search stock – "xyzs"

Now create a simple model that can hold stock data; it will look like below. Add this class to our WinForm application:

```
public class Stock
{
    public int Id { get; set; }
    public string StockName { get; set; }
    public int Volume { get; set; }
    public double Price { get; set; }
    public DateTime TradeDate { get; set; }
}
```

Now add an asynchronous method to make a call to an API and returns list of stocks, this method accepts search text entered and cancellation token to initiate cancellation if needed. This cancellation token is passed to overload of `GetAsync` method as well to cancel response from API. We will further filter data based on search text and return type of `BindingSource` which can be bound to gridview. The following is the code for this method:

```
/// <summary>
/// Async method to retrieve data from stocks API
/// </summary>
/// <param name="intputSearchtext">Search text</param>
/// <param name="ctsAPI">Cancellation token</param>
/// <returns>Binding source</returns>
private async Task<BindingSource> GetDataFromAPIAsync(string
intputSearchtext, CancellationToken ctsAPI)
{
    BindingSource bindingSource1 = new BindingSource();
    Uri requestUri = new Uri("https://localhost:44394/api/
Stocks");
    using (HttpClient client = new HttpClient())
    {
        var response = await client.GetAsync(requestUri,
ctsAPI);
        response.EnsureSuccessStatusCode();
    }
}
```

```
        var content = await response.Content.  
ReadAsStringAsync();  
  
        var data = JsonConvert.  
DeserializeObject<IEnumerable<Stock>>(content);  
        bindingSource1.DataSource = data.Where(price => price.  
StockName == intputSearchtext);  
    }  
    return bindingSource1;  
}  
  
//Create an object of type CancellationTokenSource that will be used  
to pass into our asynchronous method so that it can be used trigger  
cancellation if needed.  
CancellationTokenSource cts = null;
```

Now in the event handler of search button we will call our asynchronous method `GetDataFromAPIAsync` to retrieve stocks and pass cancellation object `cts`. Now the same **Search** button will be used to cancel while retrieving data from API so we would add small if condition based on which `cts` object is initialized, that is, for search new object is created but for cancellation we will `Cancel` method and return from the event to stop retrieving data from API. With this our `search` method will look like below:

```
/// <summary>  
/// Search stock click event handler  
/// </summary>  
/// <param name="sender"></param>  
/// <param name="e"></param>  
private async void Search_Click(object sender, EventArgs e)  
{  
    stockData.Rows.Clear();  
    stockData.Refresh();  
    var ticker = new Stopwatch();  
    ticker.Start();  
    search.Text = "Cancel";  
  
    //On clicking of Search/Cancel checking to cancel operation  
    or perform search
```

```
if (cts != null)
{
    cts.Cancel();
    cts = null;
    return;
}

this.cts = new CancellationTokenSource();

//Delegate on cancellation token when there is a
cancellation, executes on calling thread's context in this case UI
this.cts.Token.Register(() =>
{
    progressMessage.Text = "Search is cancelled" ;
});

//Cancellation needs to be handled gracefully
try
{

    var getData = await GetDataFromAPIAsync(searchText.Text,
this.cts.Token);
    stockData.DataSource = getData;
}
catch (OperationCanceledException ex)
{
    Logs.Text = ex.Message;
}
finally
{
    cts = null;
}

progressMessage.Text = $"Loaded stocks for {searchText.Text}
```

```
in {ticker.ElapsedMilliseconds}ms";
    search.Text = "Search";
}
```

Once we run this code and type some data and click on **Search**, the **Search** button text is changed to **Cancel** button as shown in *Figure 6.7*:

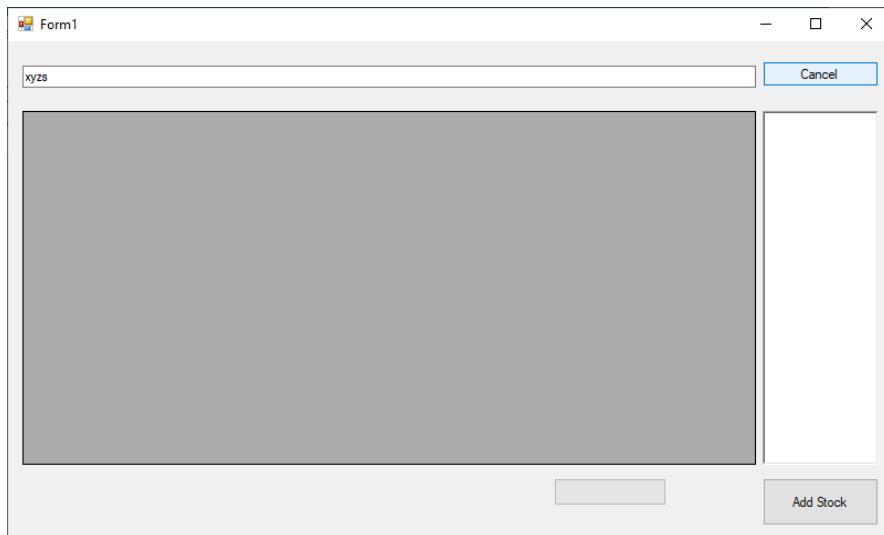


Figure 6.7: Windows form showing cancel button

Click on the **Cancel** button, and it would initiate cancellation until API and allows users to search for new data, as shown in *Figure 6.8*. Also, the cancellation message is printed in the text area:

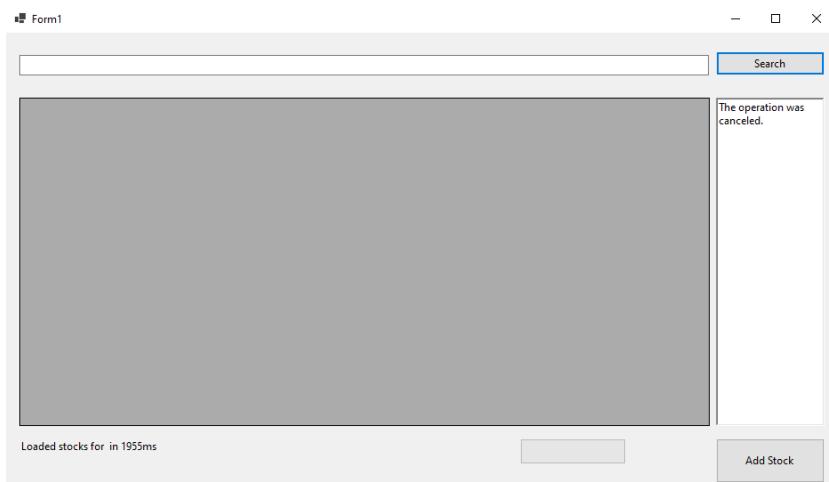


Figure 6.8: Windows form after canceling search operation

In the preceding example, the user tried to search stock xyzs; however, canceled operation immediately, and that has returned from `async` operation with `OperationCanceledException` and grid is never loaded. It gives better user experiences as it allows the user to search again if needed. Cancellation token also gives us the option to subscribe to a callback, which again runs on calling thread to perform any specific operation. In this example, it is used to update the progress text, as illustrated in the following code:

```
//Delegate on cancellation token when there is a cancellation executes
on calling thread's context in this case UI

this.cts.Token.Register(() =>
{
    progressMessage.Text = "Search is cancelled" ;
});
```

With .NET core cancellation token's call back method would be the first code that is executed after cancellation so in our code `progressMessage.Text` is updated final output is "**Loaded stocks for in...**" and that's because as soon as the cancellation is triggered remaining of the caller method's code is executed after executing `callback`.

Note: With the .NET framework, this was the other way around; that is, the first caller's remaining code is executed, and then the callback is executed.

However, there could be cases where there is a need that `async` operation doesn't throw an exception (`OperationCanceledException`) but return normally. However, in this case, the calling method needs not to handle such exception, for example, calculating prime numbers less than a huge number or reading line by line from a file; in such cases, there may need to use partial data that is received/processed. A similar example is shown below, and its output is shown in *Figure 6.9*:

```
/// <summary>
/// Async method doing high CPU operation, Add this to form
/// </summary>
/// <returns></returns>
private async Task<long> DoHighCPUIntense(CancellationToken
token)
{
    long counter = 0;
    search.Text = "Stop";
    Task<long> output = Task.Run(() =>
    {
```

```
        while (true)
    {
        counter++;
        if (token.IsCancellationRequested)
        {
            counter++;
            break;
        }
    }
    return counter;
}, token);
try
{
    await output;
}
catch (AggregateException agEx)
{
    throw agEx;
}
return counter;
}
```

Further call this method in **Search** button click:

```
highCPUCount = await DoHighCPUIntense(cts.Token);
Logs.Text += $"Counted till {highCPUCount.ToString()}" + Environment.
NewLine;
```

Running this application and clicking on **Search** button will give output as shown in *Figure 6.9*, where it retrieved data till cancellation is initiated:

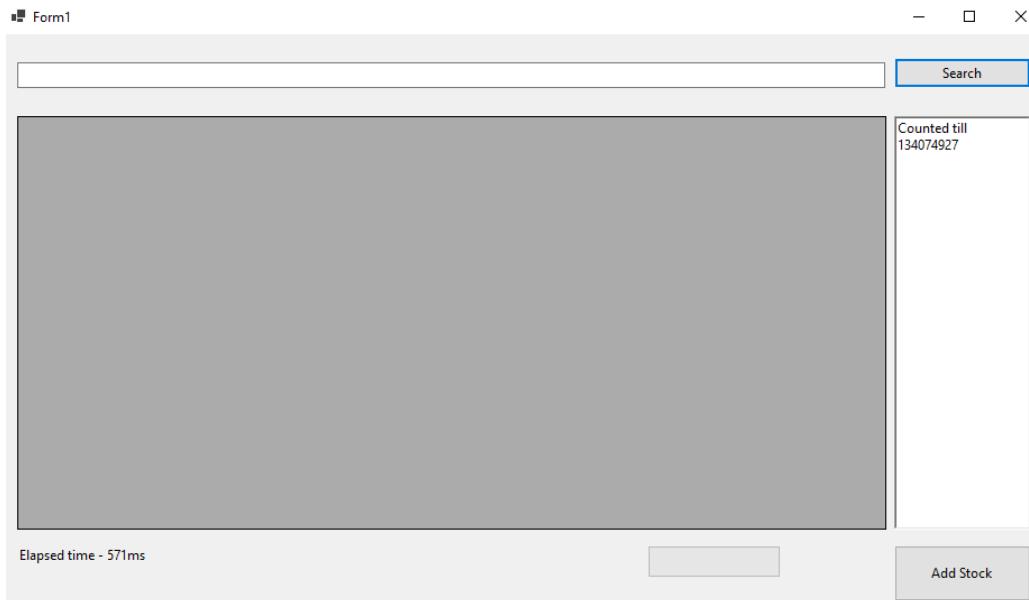


Figure 6.9: Windows form after canceling search operation and handling without exception

Cancellation in this method is validating if cancellation is requested and then returning gracefully to the caller, which further handles it. You can notice that in the following example code won't go to the `catch` block.

The cancellation also supports something called `CancellationToken.None` which tells the caller that method can never be canceled. So in the above example, we can pass `CancellationToken.None` to `GetAsync` method, which means that API call won't be canceled at the network level even though from UI search method is canceled.

Note: Please refer to sample Stocks.API project under *Chapter 6* folder for the API implementation.

Progress reporting

With `async` pattern developer can run multiple long-running operations in the background, reporting progress on how much of operation is completed/pending is a vital feature to enhance the user experience; this helps users to cancel a long-running request and gives more fluidic behavior to your application. Typically, applications will have this a progress bar and a cancel button to indicate the amount of work done/remaining and the **Cancel** button if the user wants to opt-out from a long-running operation.

With `async` methods, this can be achieved by adding a parameter of type `IProgress<T>` to the signature of the `async` method, here `T` can be any type that will hold progress information.

For example, it can be simple integer type where the amount of work completed can be reported back in percentage and serves the purpose for most of the scenarios or a complex type that holds additional information like time is taken between progressing from say 10% to 20% (will see this further in our example).

`IProgress` provides `Report(T)` method to pass progress information back to the UI thread from a background task. This information can be used by say a `progressbar` control in the WPF app to update the user on the completed / remaining work.

Let's create a simple WPF application that downloads multiple files from the web and write it to disk locally. Let us name it `FileDownload`, now add controls to it as below:

- **2 Buttons:** Name them `FileDownload`, `FileDownload1`
- **2 Textboxes:** Name them `logs`, `logs1`
- **2 Progress bars:** Name them `taskProgress`, `taskProgress1`
- **2 Labels:** Name them `totalTimeTaken`, `totalTimeTaken1`

Once we add these controls, position them, as shown in *Figure 6.10*:



Figure 6.10: File download WPF application

First, create a model to report the progress back to the caller, let's call it `ProgressReport`, and it will look like below:

```
public class ProgressReport
{
    public double progressPercentage { get; set; }
    public long totalBytes { get; set; }
    public int bytesToRead { get; set; }
    public long elapsedTime { get; set; }
}
```

Now add a `private async` method that downloads the file from the internet; in this case, we are downloading a large file from GitHub through a stream and writing the downloaded stream immediately. As mentioned, one of the parameter to the download method is of type `IProgress<T>`, so method signature supporting progress will look something like below:

```
private async Task DownloadLargeFileAsync(string fileToDownload, string
fileName, CancellationToken token, IProgress<ProgressReport> progress)
```

Here `T` is custom type as we want to pass additional information to the progress bar, so in our case, we will pass model `ProgressReport`. Now when we want to report the progress first thing that we need to know is the overall file size so that we can calculate the percentage of the file that is downloaded. So, to first receive the overall file size, we will make use of `HttpCompletionOption.ResponseHeadersRead` and pass that to the overload of the `GetAsync` method stream to further write it to a file.

With all this `DownloadLargeFileAsync` will look like below:

```
/// <summary>
/// Async Download Method
/// </summary>
/// <param name="fileToDownload">File to download</param>
/// <param name="fileName">Name of file to write locally</param>
/// <param name="token">Cancellation token</param>
/// <param name="progress">Progress reporting</param>
private async Task DownloadLargeFileAsync(string fileToDownload,
string fileName, CancellationToken token, IProgress<ProgressReport>
progress = null)
{
    var ticker = new Stopwatch();
    ticker.Start();
    byte[] buffer = new byte[8192];
```

```
int bytes = 0;
string fileToWriteTo = System.IO.Path.Combine(System.IO.Path.
GetTempPath(), fileName);
using (HttpClient client = new HttpClient())
{
    string url = fileToDownload;
    using (HttpResponseMessage response = await client.
GetAsync(url, HttpCompletionOption.ResponseHeadersRead, token))
    {
        response.EnsureSuccessStatusCode();
        long totalLength = response.Content.Headers.
ContentLength.HasValue ? response.Content.Headers.ContentLength.Value :
34632982; //Occasionally github returns response without content length
header hence in that case defaulting to actual file size
        using (Stream stream = await response.Content.
ReadAsStreamAsync(), fileStreamToWrite = new FileStream(fileToWriteTo,
FileMode.Create, FileAccess.Write, FileShare.None, 1024, true))
        {
            for (; ; )
            {
                int dataToRead = await stream.
ReadAsync(buffer, 0, buffer.Length, token);
                if (dataToRead == 0)
                {
                    break;
                }
                else
                {
                    await fileStreamToWrite.
WriteAsync(buffer, 0, dataToRead); //Writing stream to disk as and when
chunk is downloaded
                    var data = new byte[dataToRead];
                    buffer.ToList().CopyTo(0, data, 0,
dataToRead);
                    bytes += dataToRead;
                    if (progress != null) //For calling
methods that do not want to report progress
                }
            }
        }
    }
}
```

```
{  
    if (((bytes * 100) / totalLength) % 5 ==  
0) //reporting progress for every 5%  
    {  
        progress.Report(new ProgressReport()  
        {  
            progressPercentage = (bytes *  
1d) / (totalLength * 1d) * 100,  
            bytesToRead = bytes,  
            totalBytes = totalLength,  
            elapsedTime = ticker.  
ElapsedMilliseconds  
        });  
    }  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}  
}
```

Now let us add cancellation tokens to the `FileDownload` class and constant to store the name of large filename as shown in the following code:

```
CancellationTokenSource cts = null;  
CancellationTokenSource cts1 = null;  
const string fileName = "largefile.zip";
```

In calling method, that is, button click; in this case, we create an object of `Type Progress<T>` where T is of type `ProgressReport` and pass it to the `async` method, which takes care of invoking every time `progress.Report(T)` is called. `Progress<T>` is a framework class that inherits `IProgress<T>`; when we are creating an object of this class, it saves current threads (in this case UI thread) `SynchronizationContext` and each time report method is called it raises event handlers on the calling thread. As you can see, there are two ways to do it—either by the constructor or through the `ProgressChanged` event.

```
//Progress reporting
```

```
var progress = new Progress<ProgressReport>(percent =>
{
    taskProgress.Value = percent.progressPercentage;
    logs.Text += $"{percent.bytesToRead}/{percent.
totalBytes} downloaded!!{Environment.NewLine}";
    logs.Text += $"Elapsed time - {percent.elapsedTime}
ms{Environment.NewLine}";
});
```

Or the following code:

```
var progress = new Progress<ProgressReport>();
progress.ProgressChanged += (s, e) =>
{
    taskProgress.Value = e.progressPercentage;
    logs.Text += $"{e.bytesToRead}/{e.totalBytes}
downloaded!!{Environment.NewLine}";
    logs.Text += $"Elapsed time - {e.elapsedTime}
ms{Environment.NewLine}";
};
```

With all this code inside `fileDownload` button click event handler will look like below:

```
namespace Stocks.WPF
{
    /// <summary>
    /// Interaction logic for FileDownload.xaml
    /// </summary>
    public partial class FileDownload : Window
    {

        private async void FileDownload_Click(object sender,
RoutedEventArgs e)
        {
            fileDownload.Content = "Cancel";
            taskProgress.Visibility = Visibility.Visible;
```

```
taskProgress.IsIndeterminate = false;
taskProgress.Value = 0;
taskProgress.Maximum = 100;
logs.Text = "";

//On clicking of Search/Cancel checking to cancel operation
or perform search
if (cts != null)
{
    cts.Cancel();
    cts = null;
    return;
}
cts = new CancellationTokenSource();

//Progress reporting
var progress = new Progress<ProgressReport>(percent =>
{
    taskProgress.Value = percent.progressPercentage;
    logs.Text += $"{percent.bytesToRead}/{percent.
totalBytes} downloaded!!{Environment.NewLine}";
    logs.Text += $"Elapsed time - {percent.elapsedTime}
ms{Environment.NewLine}";
});

try
{
    await DownloadLargeFileAsync("https://github.com/
Ravindra-a/largefile/archive/master.zip", "largefile.zip", cts.Token,
progress);
    logs.Text += $"File {fileName} downloaded
successfully!!{Environment.NewLine}";
}
catch (OperationCanceledException ex)
{
```

```
        logs.Text = ex.Message;
    }
    catch (Exception ex)
    {
        logs.Text = ex.Message;
    }
    finally
    {
        cts = null;
        taskProgress.Visibility = Visibility.Hidden;
        fileDownload.Content = "File Download";
        totalTimeTaken.Content = "Download largefile.zip completed";
    }
}
}
}
```

We will add a similar code to the `FileDownload1` button click, and it will look exactly like the other button but using another set of controls on the screen. The code will look like below:

```
private async void FileDownload1_Click(object sender, RoutedEventArgs e)
{
    fileDownload1.Content = "Cancel";
    taskProgress1.Visibility = Visibility.Visible;
    taskProgress1.Indeterminate = false;
    taskProgress1.Value = 0;
    taskProgress1.Maximum = 100;
    logs1.Text = "";

    //On clicking of Search/Cancel checking to cancel operation
    or perform search
    if (cts1 != null)
    {
        cts1.Cancel();
    }
}
```

```
        cts1 = null;
        return;
    }

    cts1 = new CancellationTokenSource();

    //Progress reporting
    var progress = new Progress<ProgressReport>(percent =>
    {
        taskProgress1.Value = percent.progressPercentage;
        logs1.Text += $"{percent.bytesToRead}/{percent.
totalBytes} downloaded!!{Environment.NewLine}";
        logs1.Text += $"Elapsed time - {percent.elapsedTime}
ms{Environment.NewLine}";
    });

    try
    {
        await DownloadLargeFileAsync("https://github.com/
Ravindra-a/largefile/archive/master.zip", "largefile1.zip", cts1.Token,
progress);
        logs1.Text += $"File {fileName} downloaded
successfully!!{Environment.NewLine}";
    }
    catch (OperationCanceledException ex)
    {
        logs1.Text = ex.Message;
    }
    catch (Exception ex)
    {
        logs1.Text = ex.Message;
    }
    finally
    {
        cts1 = null;
    }
}
```

```

        taskProgress1.Visibility = Visibility.Hidden;
        fileDownload1.Content = "File Download";
        totalTimeTaken1.Content = "Download largefile1.zip
completed";
    }
}

```

Once we run this code and click on buttons `FileDialog` and `FileDialog1` WPF app will look like the following screenshot, where we see the download progress in the progress bar, and the text box will display bytes downloaded to total bytes. Once the download is complete there will be a message on UI confirming on completed download, and then if you navigate to Temp folder there will be a file with name `largefile.zip`:

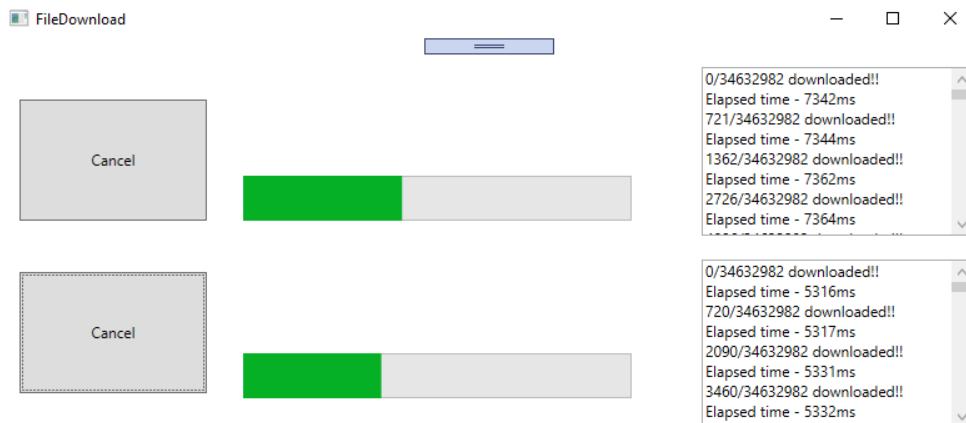


Figure 6.11: File download WPF application, reporting progress

As you can see in the example, this works well in tandem with cancellation; it is recommended to implement cancellation along with progress to provide a seamless experience for end-users. Here we saw on how easily we can build an application using a Task-based async pattern that gives a better user experience by reporting on progress using `progress<T>`, `IProgress<T>`. These principles can further be used to build much more real-world complex applications.

Note: `IProgress<T>` is not exclusive for asynchronous methods; it can be very well used in synchronous methods.

Other asynchronous patterns

Till now, we have seen how to implement an asynchronous pattern using `async-await`, however asynchronous methods existing even before `async-await`. In the next section, patterns on how asynchronous methods were implemented before `async-`

await and how to build wrappers on top of them so that they can be consumed using async-await in any modern application.

Asynchronous Programming Model (APM)

Asynchronous Programming Model (APM), also known as the `IAsyncResult` pattern, is one of the legacy patterns using which asynchronous operations can be implemented. This pattern expects the asynchronous operation to be split into two methods, one starting with `Begin` and another starting with the end, something like `BeginRead` and `EndRead` and an optional `callback` method:

- The `Begin` method is used to start asynchronous operation where the return type of such an operation should be of type `IAsyncResult`.
- The `End` method takes a parameter of type `IAsyncResult`, that is, the output of the `Begin` method. It used to indicate the completion of the `async` operation and to retrieve the result/output of the asynchronous operation.
- An optional callback is passed, which gets triggered on completion of `Begin` operation; typically, this is used to call the `End` method.

Taking an example of a typical TAP method from framework `Stream` class:

```
public Task WriteAsync(byte[] buffer, int offset, int count)
```

Corresponding APM methods look like the following code:

```
public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int count, AsyncCallback callback, object state)
public virtual void EndWrite(IAsyncResult asyncResult)
```

So, a simple file read asynchronous operation representing a file read will look like the following code:

```
class Program
{
    static Byte[] bytes = new Byte[100];
    static void Main(string[] args)
    {
        Stopwatch watch = new Stopwatch();
        watch.Start();
        FileStream fs = new FileStream(@"../../../../../TextFile.txt", FileMode.Open, FileAccess.Read, FileShare.Read, bytes.Length, FileOptions.Asynchronous);
        Console.WriteLine($"Begin reading file, Elapsed time - {watch.ElapsedMilliseconds}");
    }
}
```

```
    IAsyncResult result = fs.BeginRead(bytes, 0, bytes.Length,
null, null);

    while (!result.IsCompleted) // Proceeding with doing some
other operation while file is being read
    {

        Console.WriteLine($"Do something else in main method
while reading file, Elapsed time - {watch.ElapsedMilliseconds}");

    }

    int numBytesRead = fs.EndRead(result);

    Console.WriteLine($"End reading file, Number of bytes -
{numBytesRead}, Elapsed time - {watch.ElapsedMilliseconds}");

    fs.Close();
    watch.Stop();

    Console.WriteLine($"File contents - {Encoding.Default.
GetString(bytes)}");

    Console.ReadKey();
}

}
```

Once we run this code, it will give output, as shown in *Figure 6.12*:

```
Begin reading file, Elapsed time - 6
Do something else in main method while reading file, Elapsed time - 31
Do something else in main method while reading file, Elapsed time - 41
Do something else in main method while reading file, Elapsed time - 41
Do something else in main method while reading file, Elapsed time - 41
Do something else in main method while reading file, Elapsed time - 41
Do something else in main method while reading file, Elapsed time - 41
Do something else in main method while reading file, Elapsed time - 41
Do something else in main method while reading file, Elapsed time - 41
Do something else in main method while reading file, Elapsed time - 41
End reading file, Number of bytes - 32, Elapsed time - 41
File contents - This is awesome!!!!!!!!!!!!!!
```

Figure 6.12: Output of file read using APM pattern

Now changing the preceding operation synchronously using the following code:

```
int numBytesRead = fs.Read(bytes, 0, bytes.Length);

Console.WriteLine($"Do something else in main method while
reading file, Elapsed time - {watch.ElapsedMilliseconds}");

Console.WriteLine($"End reading file, Number of bytes -
{numBytesRead}, Elapsed time - {watch.ElapsedMilliseconds}");
```

Once we run this code, it will give output, as shown in *Figure 6.13*:

```
Begin reading file, Elapsed time - 9
Do something else in main method while reading file, Elapsed time - 93
End reading file, Number of bytes - 32, Elapsed time - 93
File contents - This is awesome!!!!!!!!!!!!!!
```

Figure 6.13: Output of file reading synchronously

Here we can see (based on the elapsed time) that in *Figure 6.13* line with the message **Do Something else..** is executed after the read operation is completed because there we are reading file synchronously, however, with APM (*Figure 6.12*) way we can parallelly do something else while the file is being read.

However, calling `EndRead` immediately after `BeginRead` won't be a realistic scenario; that's where the `Begin` method of APM needs to support optional callback operation, which gets called once the asynchronous operation is completed.

So, let's add a callback method as shown below to the console application:

```
/// Callback method
private static void EndRead(IAsyncResult asyncResult)
{
    Console.WriteLine($" Managed Thread Id in endread is :
{Thread.CurrentThread.ManagedThreadId}"); //// The managed thread
identifier.

    FileStream fs = (FileStream) asyncResult.AsyncState;
    Int32 numBytesRead = fs.EndRead(asyncResult);
    Console.WriteLine($" Number of bytes - {numBytesRead}");
    Console.WriteLine(Encoding.UTF8.GetString(bytes));
    fs.Close();
}
```

Now changing the `Main` method as shown below, additionally here we can see that the callback operation is performed on a different thread as intended:

```
class Program
{
    static Byte[] bytes = new Byte[100];
    static void Main(string[] args)
    {
        Console.WriteLine($" Managed Thread Id in Main is : {Thread.
CurrentThread.ManagedThreadId}"); //// The managed thread identifier.
```

```
        Stopwatch watch = new Stopwatch();
        watch.Start();
        FileStream fs = new FileStream(@"../../../../../TextFile.
txt", FileMode.Open, FileAccess.Read, FileShare.Read, bytes.Length,
FileOptions.Asynchronous);
        Console.WriteLine($" Begin reading file, Elapsed time -
{watch.ElapsedMilliseconds}");
        fs.BeginRead(bytes, 0, bytes.Length, EndRead, fs);
        Console.WriteLine($" Do something else in main method while
reading file, Elapsed time - {watch.ElapsedMilliseconds}");
        watch.Stop();
        Console.ReadKey();
    }
}
```

Here we are passing a callback to `BeginRead`, which takes care of parallel reading the file. Once we run this code, it will give output, as shown in *Figure 6.14*:

```
Managed Thread Id in Main is : 1
Begin reading file, Elapsed time - 14
Do something else in main method while reading file, Elapsed time - 78
Managed Thread Id in endread is : 4
Number of bytes - 32
This is awesome!!!!!!!!!!!!!!
```

Figure 6.14: Output of file read using APM pattern with callback

However, with the introduction of TAP, APM is no longer recommended, so it is good to know about APM but not recommended to use it.

APM to TAP wrapper

One use case where this can help is if there is a legacy/third party library that supports async operation using APM methods (`BeginOperation/EndOperation`), it is good to have an understanding of APM to build a wrapper that can be used to expose APM operations as TAP operations.

It can be implemented by using `Task.Factory.FromAsync` method, which takes the `Begin` and `End` method as input and returns a `Task`. For example, to read a file TAP wrapper method around its APM methods would look like the following code:

```
Task<int> ReadAsyncAPMWrapper(FileStream fs, byte[] buffer, int
offset, int count)
```

```

{
    return Task<int>.Factory.FromAsync(fs.BeginRead, fs.EndRead,
buffer, offset, count, null);
}

```

This method can be called just like any other TAP method with `await` keyword. Something like:

```

FileStream fs = new FileStream(@"../../../../../TextFile.txt", FileMode.
Open, FileAccess.Read, FileShare.Read, bytes.Length, FileOptions.
Asynchronous);

numBytesRead = await ReadAsyncAPMWrapper(fs, bytes, 0, bytes.Length,
cts.Token);

```

However, there are limitations with this implementation in terms of things like cancellation token and better exception handling, logging. So, considering the same example above, we will use `BeginRead` and `EndRead` of `FileStream` class and build a TAP wrapper on top of it by wrapping around APM methods in a `Task` and using `TaskCompletionSource` to signal completion or cancellation. The following code demonstrates how to implement a more sophisticated TAP wrapper over APM operations:

```

class Program
{
    static Byte[] bytes = new Byte[100];
    static CancellationTokenSource cts = new
CancellationTokenSource();
    static async Task Main(string[] args)
    {
        Console.WriteLine($"Managed Thread Id in Main is : {Thread.
CurrentThread.ManagedThreadId}"); //// The managed thread identifier.

        Stopwatch watch = new Stopwatch();
        watch.Start();

        FileStream fs = new FileStream(@"../../../../../TextFile.
txt", FileMode.Open, FileAccess.Read, FileShare.Read, bytes.Length,
FileOptions.Asynchronous);

        Console.Write("Enter wait time in seconds before cancelling
operation ");

        int waitTime = Convert.ToInt32(Console.ReadLine());
        cts.CancelAfter(waitTime * 1000);

        int numBytesRead = 0;
    }
}

```

```
try
{
    numBytesRead = await ReadAsyncAPMWrapper(fs, bytes, 0,
bytes.Length, cts.Token);
    Console.WriteLine("Operation completed");
}
catch (OperationCanceledException ex)
{
    Console.WriteLine($"Operation cancelled - {ex.
Message}");
}
finally
{
    cts = null;
    fs.Close();
    Console.WriteLine($"Number of bytes - {numBytesRead}");
}
Console.ReadKey();
}

/// TAP Wrapper over BeginRead and EndRead of FileStream
static Task<int> ReadAsyncAPMWrapper(FileStream fs, byte[]
buffer, int offset, int count, CancellationToken token)
{
    var taskCompletionSource = new TaskCompletionSource<int>();
    //Registering cancellation token, although this is not an
elegant way to cancel as it doesn't handle IO resource cleanly.
    // also this doesn't stop beginread
    token.Register(() => taskCompletionSource.TrySetCanceled());
    fs.BeginRead(buffer, offset, count, iAsyncResult =>
{
    try
    {
        Thread.Sleep(5000); //If user input has waited more
than this complete operation else cancel.
    }
    catch (OperationCanceledException)
    {
        taskCompletionSource.TrySetCanceled();
    }
})
```

```

        if (token.IsCancellationRequested)
        {
            throw new OperationCanceledException();
        }

        var state = iAsyncResult.AsyncState as FileStream;
        var read = state.EndRead(iAsyncResult);
        taskCompletionSource.TrySetResult(fs.EndRead(read));
    }

    catch (Exception exc)
    {
        taskCompletionSource.TrySetException(exc);
    }
}, fs);

return taskCompletionSource.Task;
}

}
}

```

Here we are using `TrySetResult` and `TrySetException` methods to complete the `Task<int>` exposed through `Task` property of `TaskCompletionSource` class. As you can see, the rest of the bit in terms of exception handling and cancellation remains the same as any other TAP method. The output of this code will look like the following screenshot:

```

Managed Thread Id in Main is : 1
Enter wait time in seconds before cancelling operation 6
Operation completed
Number of bytes - 32

```

Figure 6.15: Output TAP wrapper over APM methods

Now enter wait time less than 5 (as per if the condition that triggers cancellation) let us say 4, cancellation gets triggered, and if we see no file data is read. The output of this will look like, as shown in *Figure 6.16*:

```

Managed Thread Id in Main is : 1
Enter wait time in seconds before cancelling operation 4
Operation cancelled - A task was canceled.
Number of bytes - 0

```

Figure 6.16: Output TAP wrapper over APM methods, here job is canceled before completion

In *Summary*, what we are doing here is taking APM methods, combining it to a single method, and wrapping them in a task.

Note: Although we can cancel this operation, it's not an elegant way to do due to limitations in APM methods; in reality, we are not canceling the file read operation but only canceling wait on the operation.

TAP to APM wrapper

If we want to do vice versa, that is, converting the TAP method to APM all we need to take the task method and split it into two methods:

1. The first one that can take `AsyncCallback` and `State` of the calling object and return `IAsyncResult`
2. Second one accepting `IAsyncResult` from the previous step

To build the first method:

- We will make use of `TaskCompletionSource` class and its capability to create task instance
- Manually handle `IsFaulted`, `IsHandled` methods or the successful result of `TaskCompletionSource` task property to mirror results into task instance through `TrySetException`, `TrySetCanceled`, and `TrySetResult`
- Pass `TaskCompletionSource` task instance to callback provided (If any) and return same task instance

So, if we build a helper method on `Task`, it would look like below:

```
public static class TaskAPMExtension
{
    /// Generic extension method to convert TAP methods to APM
    public static IAsyncResult TAPToApm<TResult>(this Task<TResult>
task, AsyncCallback asyncCallback, object state)
    {
        var taskCompletionSource = new
TaskCompletionSource<TResult>(state);

        task.ContinueWith(delegate
{
    if (task.IsFaulted)
    {
```

```
taskCompletionSource.TrySetException(task.Exception.  
InnerExceptions);  
}  
else if (task.IsCanceled)  
{  
    taskCompletionSource.TrySetCanceled();  
}  
else  
{  
    taskCompletionSource.TrySetResult(task.Result);  
}  
  
if (asyncCallback != null)  
{  
    asyncCallback(taskCompletionSource.Task);  
}  
  
, CancellationToken.None, TaskContinuationOptions.None);  
  
return taskCompletionSource.Task;  
}  
}  
//With the help of this helper method any TAP async method can be  
converted into begin operation of APM.  
static IAsyncResult BeginAPI(AsyncCallback callback, object  
state)  
{  
    return GetStocksAsync(cts.Token).TAPToApm(callback, state);  
}
```

For the second method, all we need is to take the response of the first method, pass it as an input parameter, and get the result through `Result` property of `Task<TResult>`. Let's create a console application and add an asynchronous method to retrieve data from API; this method definition will look like below:

```
/// Async method to retrieve data from API
```

```
static async Task<string> GetStocksAsync(CancellationToken token)
{
    using (HttpClient client = new HttpClient())
    {
        try
        {
            var response = await client.GetAsync("https://localhost:44394/api/Stocks", token);
            response.EnsureSuccessStatusCode();
            var content = await response.Content.
ReadAsStringAsync();
            Console.WriteLine("Data retrieved from API");
            return content;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"exception occurred in API - {ex.Message}");
            throw;
        }
    }
}
```

The next thing is we need to make use of `CancellationTokenSource` and the helper method created to implement `BeginMethod`, `EndMethod`, and optional `CallbackMethod`. So, our implementation of the calling method will look like the following code along with cancellation and error handling:

```
class Program
{
    static CancellationTokenSource cts = new
CancellationTokenSource();
    static void Main(string[] args)
    {
        Stopwatch watch = new Stopwatch();
        watch.Start();
        IAsyncResult result = BeginAPI(null, null);
```

```
        while (!result.IsCompleted) // Proceeding with doing some
other operation while file is being read
    {
        if (watch.ElapsedMilliseconds % 3000 == 0)
        {
            cts.CancelAfter(15000);
            Console.WriteLine($"Do something else in main
method while receiving response from API, Elapsed time - {watch.
ElapsedMilliseconds}");
        }
    }

    string apiResponse = EndAPI(result);
    Console.WriteLine($"API response - {apiResponse}, Elapsed
time - {watch.ElapsedMilliseconds}");

    //IAsyncResult result = BeginAPI(EndAPIUsingCallback, null);
// Using callback
    Console.ReadKey();
    watch.Stop();
}

/// Begin operation
static IAsyncResult BeginAPI(AsyncCallback callback, object
state)
{
    return GetStocksAsync(cts.Token).TAPToApm(callback, state);
}

/// End operation
static string EndAPI(IAsyncResult asyncResult)
{
    try
    {
        return ((Task<string>)asyncResult).Result;
    }
    catch (AggregateException ex)
```

```
{  
    List<Tuple<string, string>> errors = ex.Flatten().  
    InnerExceptions.Select(x => new Tuple<string, string>(x.Message,  
    x.StackTrace)).ToList();  
  
    int counter = 0;  
  
    foreach (Tuple<string, string> error in errors)  
    {  
        counter++;  
        Console.WriteLine($"{counter}).Error - {error.Item1}  
    \n Innerstack \n {error.Item2} \n");  
    }  
  
    return $"Exception occured - {ex.Message}";  
}  
  
}  
  
/// Callback  
static void EndAPIUsingCallback(IAsyncResult asyncResult)  
{  
    string apiResponse = ((Task<string>)asyncResult).Result;  
    Console.WriteLine($"API response - {apiResponse}");  
}  
}
```

As you can see, we have implemented both variants one with callback and another without, of course, there is more we could do here, but this is a good starting point if the need arises to write an APM wrapper over TAP methods.

Event-based Asynchronous Pattern (EAP)

Event-based Asynchronous pattern is another way to add an asynchronous capability to your methods, this pattern is introduced in .NET 2.0, and the way we achieve asynchronous functionality is by splitting the method into two parts:

1. Creating a method typically suffixed with `async` and is executed on a different thread
2. An event confirming completion of `async` operation typically named with a suffix `Completed`

Additional methods that can be optionally added to `async` implementation is:

- **Cancellation support:** To cancel the `async` operation
- **Progress support:** To track the progress of completion, typically named as `Progresschanged` event has an argument of type `ProgressChangedEventArgs`
- **Returning incremental results:** A capability to return results that are received say, for example, reading file line by line and then copying it to new file immediately instead of waiting for full file download
- **IsBusyProperty:** If multiple invocations of the method are not supported implement `IsBusy` property for the caller to signal state of the previous invocation if our class supports multiple/parallel invocations of `async` operation `IsBusy` property shouldn't be implemented

Let's take an example of a class that is used to calculate pi up to N number of places, the first thing that we need to define a delegate for completion operation:

```
public delegate void CalculatePiCompletedEventHandler(object sender,
CalculatePiCompletedEventArgs e);
```

Now create a class derived from `AsyncCompletedEventArgs` which will be input to the delegate, here we can add additional properties that are needed like the caller object, time taken, and so on. It will look something like the following code in our example:

```
public class CalculatePiCompletedEventArgs : AsyncCompletedEventArgs
{
    //Additional properties, they should be read only
    public string Result { get; private set; }
    public long TimeTaken { get; private set; }
    public Object Sender { get; private set; }
    public CalculatePiCompletedEventArgs(Exception e, bool canceled,
object state) : base(e, canceled, state)
    {
        this.Result = value;
        this.TimeTaken = TimeTaken;
        this.Sender = sender;
    }
}
```

Create a class `Calculatepi` and add `async` operation completion event handler that will be triggered post completion of `async` operation:

```
public event CalculatePiCompletedEventHandler CalculatepiCompleted;
```

Secondly, create an object of the type that can raise the completion event through a callback, and then this callback needs to be executed on a thread from the thread pool so that it can dispatch the message to the synchronization context. Object of class `SendOrPostCallback` (<https://docs.microsoft.com/en-us/dotnet/api/system.threading.sendorpostcallback?view=netcore-3.1>) fulfil this requirement. The callback method needs to have a signature of accepting a single parameter of type `object` so that we can pass the state. Wiring of the callback to `SendOrPostCallback` (<https://docs.microsoft.com/en-us/dotnet/api/system.threading.sendorpostcallback?view=netcore-3.1>) delegate can be done through the constructor. This code will look like below:

```
//Delegate to handle callback and raise completion event
private SendOrPostCallback onCompletedDelegate;

public Calculatepi()
{
    onCompletedDelegate = new
SendOrPostCallback(CalculationCompleted);
}

private void CalculationCompleted(object operationState)
{
    CalculatepiCompletedEventArgs e = operationState as
CalculatepiCompletedEventArgs;

    if (CalculatepiCompleted != null)
    {
        CalculatepiCompleted(this, e);
    }
}
```

Now add the primary method that calculates pi and raises callback through the object of `CalculatePiCompletedEventArgs`; let call this method `CalculatepiValue`. This method will accept two parameters:

1. Number of steps for pi calculation
2. The object of type `AsyncOperation` which is used to track the sender state, for example in case of button click this can be used to track which button is clicked, and many more

Once the calculation of pi is completed, we will raise callback through the object of `CalculatePiCompletedEventArgs` by posting it to the caller through `AsyncOperation`. With this, the code will look like below:

```
//Dictionary to handle multiple tasks invocation, uniquely
identify each operation as one dictionary element

private HybridDictionary parallelTasks = new HybridDictionary();

void Calculatepivalue(int numsteps, AsyncOperation asyncOp)
{
    Stopwatch timer = new Stopwatch();
    timer.Start();
    numsteps++;

    //Variables used in calculation of Pi
    uint[] value = new uint[numsteps * 10 / 3 + 2];
    uint[] rem = new uint[numsteps * 10 / 3 + 2];
    uint[] pi = new uint[numsteps];

    for (int j = 0; j < value.Length; j++)
        value[j] = 20;

    //Simple looping logic to calculate Pi till the number of
    characters passed as input
    for (int i = 0; i < numsteps; i++)
    {
        uint carryForward = 0;
        for (int j = 0; j < value.Length; j++)
        {
            uint number = (uint)(value.Length - j - 1);
            uint pow = number * 2 + 1;
            value[j] += carryForward;
            uint quotient = value[j] / pow;
            rem[j] = value[j] % pow;
            carryForward = quotient * number;
        }
    }
}
```

```
        pi[i] = (value[value.Length - 1] / 10);
        rem[value.Length - 1] = value[value.Length - 1] % 10;
        for (int j = 0; j < value.Length; j++)
            value[j] = rem[j] * 10;
    }

    var result = "";
    uint c = 0;

    for (int i = pi.Length - 1; i >= 0; i--)
    {
        pi[i] += c;
        c = pi[i] / 10;
        result = (pi[i] % 10).ToString() + result;
        Thread.Sleep(10);
    }
    result = result.Substring(0, 1) + "." + result.Substring(1,
result.Length - 1);

    lock (parallelTasks.SyncRoot)
    {
        parallelTasks.Remove(asyncOp.UserSuppliedState);
    }

    //raise callback
    CalculatepiCompletedEventArgs e = new
CalculatepiCompletedEventArgs(result, timer.ElapsedMilliseconds, null,
null, false, asyncOp.UserSuppliedState);
    asyncOp.PostOperationCompleted(onCompletedDelegate, e);
    timer.Stop();
}
}
```

At this point, we can define our `async` method that accepts any specific input required for our operation and another parameter of type `object`, which will be

used to accept input user state, this parameter is used to identify completion during multiple invocations.

Execution of the synchronous method can be done through the delegate and using a callback to raise the completion event. So, we can define a delegate that matches the signature of our synchronous operation in this case, `void CalculatePi(int numsteps, AsyncOperation asyncOp)`. So, the delegate will look like below, and the `async` method will look like below:

```
private delegate void CalculationEventHandler(int numSteps,
AsyncOperation asyncOp);

public void CalculatepiAsync(int numsteps, object
operationState)
{
    AsyncOperation asyncOp = AsyncOperationManager.
CreateOperation(operationState);

    //Locking for thread safety
    lock (parallelTasks.SyncRoot)
    {
        if (parallelTasks.Contains(operationState))
        {
            throw new ArgumentException("User state parameter
must be unique", "userState");
        }

        parallelTasks[operationState] = asyncOp;
    }

    CalculationEventHandler worker = new
CalculationEventHandler(Calculatepivalue);

    //Execute process Asynchronously
    Task.Run(() => worker(numsteps, asyncOp));
}

} Now consuming it in a console application:

class Program
{
```

```

    static void Main(string[] args)
    {
        Calculatepi pi = new Calculatepi();
        pi.CalculatepiCompleted += new
CalculatepiCompletedEventHandler(Pi_CalculatePiCompleted);
        Console.WriteLine("Calculating pi to 1000 places started");
        pi.CalculatepiAsync(1000, 1000);
        Console.WriteLine("Calculating pi to 900 places started");
        pi.CalculatepiAsync(900, 900);
        Console.WriteLine("do something else");
        Console.ReadKey();
    }

    static void Pi_CalculatePiCompleted(object sender,
CalculatepiCompletedEventArgs e)
    {
        Console.WriteLine($"Calculated pi to {e.UserState.
ToString()} places - {e.Result}, time taken is {e.TimeTaken.ToString()} Milliseconds");
    }
}

```

Once we run this console application, we will see the output, as shown in *Figure 6.17*:

```

Calculating pi to 1000 places started
Calculating pi to 900 places started
do something else
Calculated pi to 900 places - 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348
253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109
75665933446128475648233786783165271201909145648566923460348610454326482133936072602491412737245870066063155881748815209
209628292540917153643678925903600113305305488204665213841469519415116694330572703657595919530921861173819326117931051185
480744623799627495673518857527248912279381830119491298336733624406566430860213949463952247371907021798609437027705392171
762931767523846748184676694051320005681271452635608277857713427577896091736371787214684409012249534301465495853710507922
79689258923542019956112129021960864834418159813629774771309966518707211349999983729780499510597317328160963185950244594
55346908302642522308253344685035261931188171010003137838752886587533208381420617177669147303, time taken is 9813 Millise
conds
Calculated pi to 1000 places - 3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034
825342117067982148086513282306647093844609550582231725359408128481117450284102701938521105559644622948954930381964428810
97566593344612847564823378678316527120190914564856692346034861045432648213393607260249141273724587006606315588174881520
920962829254091715364367892590360011330530548820466521384146951941511669433057270365759591953092186117381932611793105118
548074462379962749567351885752724891227938183011949129833673362440656643086021394946395224737190702179860943702770539217
176293176752384674818467669405132000568127145263560827785771342757789609173637178721468440901224953430146549585371050792
279689258923542019956112129021960864834418159813629774771309966051870721134999998372978049951059731732816096318595024459
455346908302642522308253344685035261931188171010003137838752886587533208381420617177669147303598253490428755468731159562
8638823537875937519577818577805321712268066130019278766111959092164201989, time taken is 10928 Milliseconds

```

Figure 6.17: Output calculate pi using EAP

Here we can see that the first three lines are printed immediately to output; however, `Pi_CalculatePiCompleted` has raised post completion of pi calculation. Also, till the time event is raised, any other operation on the first thread can be performed.

EAP to TAP wrapper

Like TAP wrapper over APM wrapper asynchronous operation implemented using EAP can be implemented using `TaskCompletionSource` class. `SetException`, `SetCanceled`, `SetResult` methods of `TaskCompletionSource` class needs to be handled based on the exception, cancellation, or successful completion of the operation, finally `Task` property of `TaskCompletionSource` class is used to return from the wrapper method.

Taking the above example first let's create a WPF application add `Calculatepi` class to that application and update `Calculatepi` to implement cancellation:

- Added a cancellation method to `Calculatepi` class where we need to remove the canceled task from our dictionary
- Call this method to break pi calculation (called it in one of the for loop to stop calculating pi) and raise completion event and pass canceled status flag to `CalculatepiCompletedEventArgs`:

```
// This method cancels a pending asynchronous operation.
public void CancelAsync(object operationState)
{
    AsyncOperation asyncOp =
parallelTasks[operationState] as AsyncOperation;
    if (asyncOp != null)
    {
        lock (parallelTasks.SyncRoot)
        {
            parallelTasks.Remove(operationState);
        }
    }
}

//Utility method to check the task status
private bool TaskCanceled(object operationState)
{
    return (parallelTasks[operationState] == null);
```

Now this cancellation logic can be added anywhere in the `Calculatepi` method, the optimum place would be the loop that iterated several steps, so let's tweak that loop as shown below and update `Calculatepi` method by deleting existing for loop and adding loop below:

```
for (int i = 0; i < numsteps; i++)
{
    //Cancel and exit if cancellation is triggered
    Thread.Sleep(5);
    if (TaskCanceled(asyncOp.UserSuppliedState))
    {
        break;
    }
    uint carryForward = 0;
    for (int j = 0; j < value.Length; j++)
    {
        uint number = (uint)(value.Length - j - 1);
        uint pow = number * 2 + 1;
        value[j] += carryForward;
        uint quotient = value[j] / pow;
        rem[j] = value[j] % pow;
        carryForward = quotient * number;
    }

    pi[i] = (value[value.Length - 1] / 10);
    rem[value.Length - 1] = value[value.Length - 1] % 10; ;

    for (int j = 0; j < value.Length; j++)
        value[j] = rem[j] * 10;
}
```

The next thing we need to implement is the wrapper method using `CalculatepiAsync` and `CalculationEventHandler` of our `Calculatepi` class. It will look something like below:

```
private Task<CalculatepiCompletedEventArgs>
TAPWrappertoAPMAsync(Calculatepi calculatepi, int numsteps, object
sender, object operationState, CancellationToken token)
{
```

```

        var tcs = new
TaskCompletionSource<CalculatepiCompletedEventArgs>();
        //Delegate on cancellation token when there is a
cancellation, executes on calling thread's context in this case UI
        token.Register(() =>
{
    calculatepi.CancelAsync(operationState);
});

calculatepi.CalculatepiCompleted += (_, e) =>
{
    if (e.Cancelled)
        tcs.TrySetCanceled();
    else if (e.Error != null)
        tcs.TrySetException(e.Error);
    else
        tcs.TrySetResult(e);
};
// Register for the event and start the operation.
calculatepi.CalculatepiAsync(numsteps, sender,
operationState);
return tcs.Task;
}

```

Code is pretty much self-explanatory, where we are making use of `TaskCompletionSource` and a handler of type `CalculatepiCompletedEvent Handler` to set the appropriate status of `TaskCompletionSource` and then finally return `Task`. As you can see, this can be further extended to an extension or a generic method.

Now call this wrapper method on a button click of a WPF application as shown in following code:

```

public partial class MainWindow : Window
{
    CalculatepiCompletedEventArgs calculatepiCompletedEventArgs = null;
    CancellationTokenSource cts = null;

    private async void Calculatepi1000TAP_Click(object sender,

```

```
RoutedEventArgs e)
{
    Calculatepi calculatepi = new Calculatepi();
    calculatepi1000TAP.Content = "Cancel";
    //On clicking of Cancel checking to cancel operation
    if (cts != null)
    {
        cts.Cancel();
        cts = null;
        return;
    }

    cts = new CancellationTokenSource();

    //Cancellation needs to be handled gracefully
    try
    {
        calculatepiCompletedEventArgs = await
TAPWrapertoAPMAsync(calculatepi, 1000, sender, "Calculate pi (1000)
TAP", cts.Token);

        output.Text += $"{calculatepiCompletedEventArgs.UserState.
ToString()} - {calculatepiCompletedEventArgs.Result}, time
taken is {calculatepiCompletedEventArgs.TimeTaken.ToString()}
Milliseconds{Environment.NewLine}";

    }
    catch (OperationCanceledException)
    {
        output.Text += $"Calculate pi(1000) TAP is cancelled";
    }
    finally
    {
        cts = null;
        calculatepiCompletedEventArgs = null;
    }
    calculatepi1000TAP.Content = "Calculate pi(1000) TAP";
}
}
```

Once we run this code and click on the button output will be like, as shown in *Figure 6.18*:

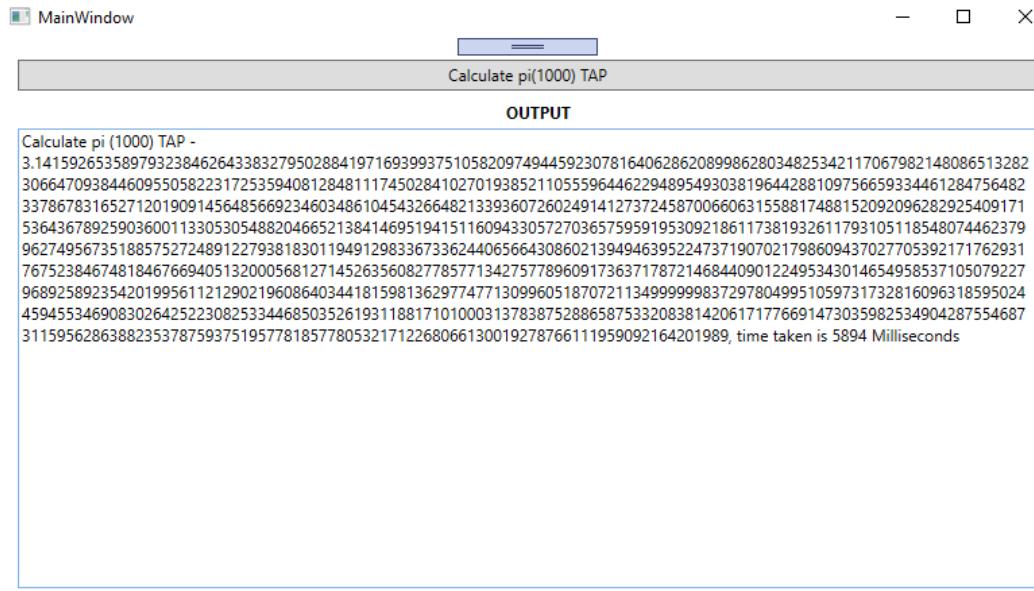


Figure 6.18: Output calculate pi using TAP wrapper over EAP

As cancellation is also implemented clicking on the same button will cancel the calculation as shown in *Figure 6.19*:

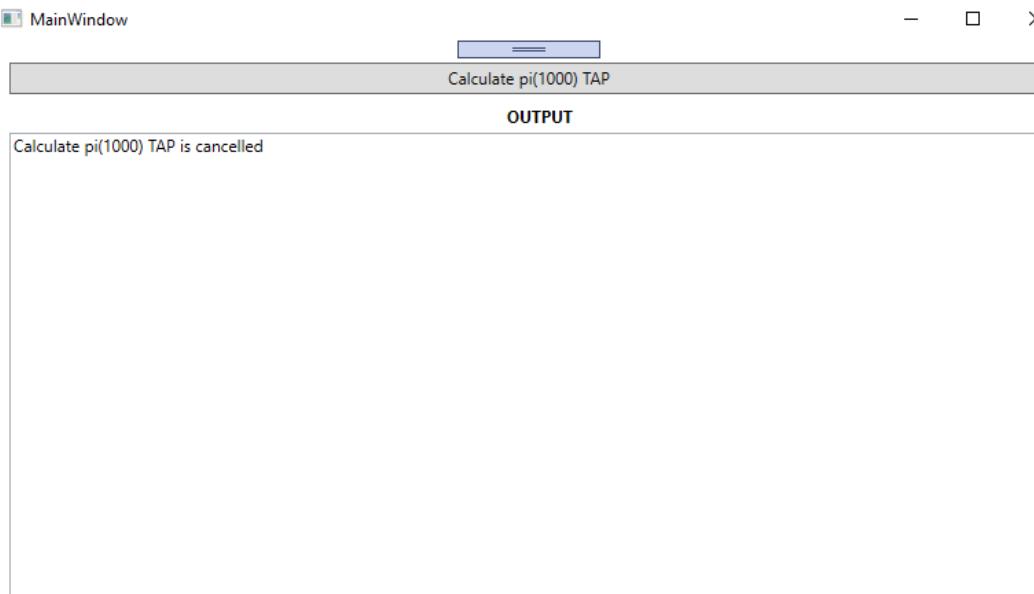


Figure 6.19: Output canceling pi calculation using TAP wrapper over EAP

So, in this section, we have implemented a wrapper method of TAP that wraps asynchronous operation that is internally implemented through APM. As we can see that in both the wrappers of TAP to EAP/APM we have made use of `TaskCompletionSource`, so the strategy remains same across where we make use of `TaskCompletionSource` and handle `SetException`, `SetCanceled`, `SetResult` methods of `TaskCompletionSource` class based on the exception, cancellation or successful completion of the operation

Note: The above example is further extended in samples to have multiple buttons that can be clicked without blocking UI and calculate pi directly through APM and TAP wrapper.

Summary

In this chapter, we have seen various patterns that can be used to implement asynchronous operations; we have seen:

- Task-based Asynchronous Pattern (TAP) in detail:
 - Exception handling
 - Cancellation
 - Report progress
 - And legacy pattern
- Event-based Asynchronous Pattern (EAP)
- Asynchronous programming model (APM)

We finished this chapter on how to implement legacy patterns and TAP wrappers over the legacy patterns, which are helpful in the case of external libraries that uses legacy patterns.

With this understanding, readers can successfully implement asynchronous methods in enterprise applications, identify areas of the application that can be implemented asynchronously and implement them accordingly.

As mentioned, if it's a new implementation, we should go with TAP; however, it is good to know EAP and APM as it would help to implement TAP wrappers on these patterns. With the understanding of these patterns, you can identify areas in your application that can be parallelized efficiently.

In the next chapter, we will see various the synchronization constructs available in .NET Core 3.0 will cover things like why is synchronization needed, different synchronization constructs/primitives, and sample programs to see synchronization constructs in action.

Exercise

1. Why is `async void` ok for the event handler?
2. Change the win forms application in the cancellation section to retrieve partial data from API, that is, retrieve data till cancellation is triggered.
3. Using Progress in TAP creates a sample to pause and resume file/data download.
4. See if you can close the stream in case of cancellation while implementing APM to TAP wrapper
5. Convert console application of calculating API into a WPF app and implement progress.

CHAPTER 7

Synchronization Constructs

"In general, good tools for staying in sync just haven't been built and made available to the world"

— Justin Rosenstein

When we are developing an application that will be executed synchronously, shared resources can be used with minimal handling as we know that at any point in time, only one thread is executing our code. We don't need any specific handling for in-memory variables because they are not shared across multiple threads. However, if our application supports executing a piece of code concurrently / parallel, handling shared resources should be one of the top priorities, not doing so would result in abnormal results. We will run into situations like deadlocks, race conditions, or getting different outputs for the same piece of code executed at different times. To solve this, we need to additionally implement something very commonly known as thread synchronization or what is also known as thread safety. In this chapter, we will see what thread synchronization is, why we need it, what can go wrong when parallel threads are not synchronized. .NET provides various synchronization constructs like locks, Semaphores, Mutex, and, many more, we will further look at how these can be implemented to achieve thread synchronization.

Structure

- Overview (Why synchronization)
- Locking constructs
 - Exclusive
 - Non-exclusive
- Signaling constructs
- Other synchronization classes
- Summary

Objectives

By the end of this chapter reader should be able to understand:

- Why we need synchronization?
- What are the exclusive locks/non-exclusive locks?
- Thread signaling
- How to use constructs like Mutex, Semaphores, AutoResetEvent, ManualResetEvent and Barrier

Overview

When we design a highly scalable application accessed by many concurrent users, there is a high possibility that the same data is read/write by multiple users at the same time. If write operation on the shared data across threads is not handled correctly (synchronously), it will lead to unexpected output. Let's see this with an example of transactions in a bank account:

- Initial amount in bank account 1000 units
- A withdraw request of 500 units is placed through an ATM
- Same time another withdraw request of 600 units are placed through internet banking

Assuming both transactions are initiated precisely at the same time, both would see a balance of 1000 units and will allow both the transactions to pass successfully, however, this will lead to an inconsistent state with data. If handled, one of the steps should fail with an exception like *Insufficient balance*. This handling of data across threads is done using synchronization and will help to get a predictable outcome. Let's see this with an example in which we add money to the bank account through multiple concurrent transactions. We start creating `BankAccount` class and add methods to increase available balance, which will start with creating class and two

private variables `BankAccount` and `numberOfTransactions`. The following code will show the implementation of the `BankAccount` class:

```
public class BankAccount
{
    private long accountBalance;
    private int numberOfTransactions;

    public int NumberOfTransactions
    {
        get
        {
            return numberOfTransactions;
        }
    }

    public BankAccount(long initialAccountBalance)
    {
        this.accountBalance = initialAccountBalance;
        numberOfTransactions = 0;
    }

    public long ShowBalance()
    {
        return this.accountBalance;
    }
}
```

Now add a private method `AddBalanceToAcccount` as below to `BankAccount` class that takes amount as a parameter and increments account Balance and `numberOfTransactions`:

```
async Task AddBalanceToAcccount(long amount)
{
    await Task.Delay(1);
    accountBalance = accountBalance + amount;
```

```
    numberOfTransactions = numberOfTransactions + 1;  
}
```

Create another public `async` method `AddMoneyToAccountAsync` which will run a loop and call `AddBalanceToAcccount`, basically what we are doing here is parallel simulating 50 transactions. The `AddMoneyToAccountAsync` method will look like the following code:

```
/// <summary>  
/// Add money to account through multiple transactions  
/// </summary>  
public async Task AddMoneyToAccountAsync()  
{  
    var tasks = new Task[50];  
    for (int i = 1; i <= tasks.Length; i++)  
    {  
        tasks[i - 1] = AddBalanceToAcccount(i);  
    }  
    await Task.WhenAll(tasks);  
}
```

Calling it through a console application expected value of variable `accountBalance` for 50 iterations should be 1275:

```
static async Task Main(string[] args)  
{  
    BankAccount bankAccount = new BankAccount(0);  
    Console.WriteLine($"Initial Balance {bankAccount.  
ShowBalance()}");  
    await bankAccount.AddMoneyToAccountAsync();  
    Console.WriteLine($"Current Balance {bankAccount.  
ShowBalance()}, total number of transactions - {bankAccount.  
NumberOfTransactions}");  
    Console.Read();  
}
```

Once we run this application output will look like the following screenshot:

```
Initial Balance 0
Current Balance 1226, total number of transactions - 47
```

Figure 7.1: Output of application without synchronization

We can see that it's lesser than what is expected, and in reality, what has happened here is since multiple threads are parallel accessing the same variable at the same time. There is no restriction on overwriting values, and at some point, few of the threads have overwritten value of variable `accountBalance` and hence unpredicted outcome. The same has happened with variable `numberOfTransactions`.

To overcome this, we need a mechanism to stop multiple threads parallel accessing shared resources, which is what synchronization is about. Hence to fix the above code, we can use one of the synchronization constructs, in this case, locks. With that implemented at any given point in time, only one thread can access the resources. In other words, only one thread can enter the critical section, and all other threads that need access to the critical section shall wait till lock is released by owning thread.

So, we create a locking object and lock critical section using that as and our method will look like this:

```
//Lock
object locker = new object();

async Task AddBalanceToAcccount(long amount)
{
    await Task.Delay(1);
    lock (locker)
    {
        accountBalance = accountBalance + amount;
        numberOfTransactions = numberOfTransactions + 1;
    }
}
```

Once synchronization is implemented using a lock, here is the output of the sample:

```
Initial Balance 0
Current Balance 1275, total number of transactions - 50
```

Figure 7.2: Output of application with synchronization

As you see, the output is what was predicted; we can see that if synchronization is not implemented for a shared resource in a multi-thread environment, there is a high possibility of data getting corrupted and that's when it becomes critical that we implement proper synchronization constructs to achieve predictable results.

Thread safety

A piece of code or method is considered as thread-safe if there aren't any resources / variables in that code that are shared across multiple threads like the first example in this chapter where current balance and number of transactions are updated at the same time by multiple threads, and it isn't thread-safe. To achieve thread-safety on a method or variable, we can either reduce the interaction between threads, that is, probably changing the application to run synchronously or using the locking / synchronization mechanism that we are going to see in the next sections.

Synchronization can be achieved through various constructs provided by .NET:

1. Locking constructs
 1. Exclusive
 2. Non-exclusive
2. Signaling constructs
3. Other synchronization classes

We will deep dive into each of these in the next sections.

Locking constructs

Locking constructs are types in .NET that help in synchronization for a shared resource between threads or coordinating insert/updates/overwrites among threads. They are primarily categorized into the following:

- **Exclusive:** Exclusive locks are the types which allow locking a resource and resource cannot be modified until the lock is released, while an object is exclusively locked no other thread can read/update that object. Exclusive locks are always acquired by one single thread at any point in and all other threads must wait till the acquiring thread release the lock. Exclusive locks are supported in .NET through:
 - Lock (`Monitor.Enter/Monitor.Exit`)
 - Mutex
 - SpinLock
- **Non-exclusive locks:** These are the types that allow the limited number of threads to access a shared resource; that is, if ten threads are trying to access a resource using a non-exclusive lock, shared resource access can be restricted

to say five threads. Usually, it is like multiple reads can be performed; however, shared resources cannot be modified until the read lock is released. .NET supports non-exclusive locks through:

- o Semaphore (Non-exclusive)
- o SemaphoreSlim (Non-exclusive)
- o Reader/Writer locks (Non-exclusive)

Taking an analogy here:

Say there is a travel website that allows you to book a seat in train with a limited capacity of 50; however, the seat is allotted inside the train. Here when we book a ticket, it is guaranteed that we will get a seat, however since the maximum number of seats is 50, precisely 50 people (threads) would be allowed to book a seat, which means 50 people have a non-exclusive lock. Once a person exits from the train entry allowed for people on the waiting list. Now inside the train, the seat occupied by a person cannot be shared, which means an exclusive lock is applied on the seat. The seat cannot be used until the person releases it.

Lock or Monitor.Enter/Monitor.Exit (Exclusive)

Lock statement is the easiest way to achieve synchronization in multi-threaded code where any shared resource within the scope of lock can be accessed using only one thread at the point in time. To lock a shared resource using `lock` statement, we need to create an object and wrap it inside `lock` keyword just like the following code:

```
object locker = new object(); //Declare lock object
async Task AddBalanceToAcccount(long amount)
{
    await Task.Delay(1);
    lock (locker) //Locking accountBalance variable
    {
        accountBalance = accountBalance + 10;
        Console.WriteLine("balance updated");
    } //Un-Locking accountBalance variable
}
```

In this example, if multiple threads parallelly call `AddBalanceToAcccount`, only one thread is allowed to access code block inside `lock` statement so only one thread can modify variable `accountBalance` at any point in time based on first come first serve basis. All the other threads will continue to wait until the lock is released by the thread that acquired it, what this means no matter the number of threads parallel

call `AddBalanceToAcccount` method, code from `lock(locker)` will always execute sequentially hence preventing data corruption.

Lock statement is in-fact syntactic sugar for `Monitor.Enter` and `Monitor.Exit` so here's how compiler converts preceding code:

```
bool lockAcquired = false;
try
{
    Monitor.Enter(locker, ref lockAcquired);
    accountBalance = accountBalance + amount;
    numberOfTransactions = numberOfTransactions + 1;
}
finally
{
    if (lockAcquired)
    {
        Monitor.Exit(locker);
    }
}
```

The output will remain the same in either case, and it is upto the developer to use whichever syntax they are comfortable with. However, for advanced thread coordination, the `Monitor` class is helpful as it has other methods like `Monitor.Wait`/`Monitor.Pulse`/`Monitor.PulseAll` that can be used for signaling. Of course, these methods can be used in tandem with the lock, but using the same construct across makes it more readable. Certain things need to be remembered for using locks:

- **We should always lock on a reference type:** The reason behind that is since `Enter` method expects an object and if a value type is passed to it boxing will occur which will create a copy of the type passed and hence when `Exit` method is called it will be a different copy again which means that they are operating on different objects. If we change `locker` to a value type like `int`, we will get a run time exception - `System.Threading.SynchronizationLockException: 'Object synchronization method was called from an unsynchronized block of code.'`
- Double-check acquiring lock as it helps in improving performance, especially in cases where code block inside lock needs to be executed only once. For example, the Singleton class or any instantiation code needs to occur if the object is `null`.

- Exception handling in locks is nothing different than a typical try...catch block in calling method; unhandled exceptions must be handled through a try...catch block or less any exception within the code block of a lock can cause the application to crash.

One last point is to avoid locks if possible, as such locking is not time-consuming or going to degrade performance; however, pausing threads and then resuming do results in some lag. So, unless and necessary avoid locks, there are types available in .NET that can be used instead of using locks like instead of Dictionary use ConcurrentDictionary.

Mutex (Exclusive)

Mutex is just like a lock (full form mutually exclusive lock); however, the scope of locking spawns across processes; that is, if multiple instances of the same process are running, a mutex can be used to execute a code block by a single thread across processes. In .NET mutex can be created by creating an object of `System.Threading.Mutex` class, the following example, will show how to create and use a mutex to achieve synchronization.

This example is a simple file create (or file upload class) class where we are writing a file to a disk, So, we will create a class called and add a method `WriteTextAsync` that takes a filename as input and writes some data into that file. Class and method implementation will look like the following code:

```
public class FileUpload
{
    private async Task WriteTextAsync(string fileName)
    {
        string text = $"Mutex is just like lock (full form mutually
exclusive lock), however scope of locking spawns across processes i.e. “
+
“if multiple instances of same process running mutex can be used to
execute a code block by a single thread across processes.”;
        byte[] encoding = Encoding.Unicode.GetBytes(text);
        await Task.Delay(1);
        using (var mutex = new Mutex(false, fileName))
        {
            mutex.WaitOne();
            using (FileStream fs = new FileStream(fileName, FileMode.
Append, FileAccess.Write, FileShare.None, bufferSize: 64, useAsync:
true))
```

```
        {
    fs.Write(encoding, 0, encoding.Length);
        }
mutex.ReleaseMutex();
}

}
```

Now we will call this method through another `async` method that will simulate parallel calls through tasks. That method will look like the following code and will be added as a `public` method in the class:

```
public async Task CreateorUpdateFiles()
{
    var tasks = new Task[50];
    for (int i = 1; i <= tasks.Length; i++)
    {
        tasks[i - 1] = WriteTextAsync($"File{i % 5}.txt");
    }

    Stopwatch timer = new Stopwatch();
    timer.Start();
    await Task.WhenAll(tasks);
    Console.WriteLine($"Time elapsed {timer.ElapsedMilliseconds}");
}
```

So, we are simulating 50 parallel calls in this method, and after every fifth iteration application writes into the same file, this method also has a timer to calculate the time taken for this operation. Now we will use this class in the main method of a simple console application, so create a console application and this class to that console application. Create an object of class `FileUpload` and call the `CreateorUpdateFiles` method. Our main method will look like the following code:

```
static async Task Main(string[] args)
{
    Console.WriteLine("Writing file to disk");
    FileUploadfileupload = new FileUpload();
```

```

        await fileupload.CreateOrUpdateFiles();
        Console.WriteLine("Writing file to disk completed");
        Console.Read();
    }

```

Once we run this application, we can see five files getting created, and each will have the text ten times (as we are looping for 50 times and writing to the same file after every 5th iteration). The output is shown in the following screenshot:

```

Writing file to disk
Time elapsed 766
Writing file to disk completed

```

Figure 7.3: Output of FileUpload application with synchronization using Mutex

If we go to the Debug folder, we can see five files are created as shown in *Figure 7.4*, and the content of the file would be the string that we passed:

Name
File0.txt
File1.txt
File2.txt
File3.txt
File4.txt

Figure 7.4: Files created in debug folder

We can see that there is no loss of data; that is, each file has ten copies of the string that we passed, and there is no run time exception. To see the benefit of Mutex, let's remove the mutex and run the application, our `WriteTextAsync` will look like the following code:

```

private async Task WriteTextAsync(string fileName)
{
    string text = $"Mutex is just like lock (full form mutually
exclusive lock), however scope of locking spawns across processes i.e. “ +
    “if multiple instances of same process running mutex can be used to
execute a code block by a single thread across processes.”;

```

```
byte[] encoding = Encoding.Unicode.GetBytes(text);
await Task.Delay(1);
    using (FileStream fs = new FileStream(fileName, FileMode.Append, FileAccess.Write, FileShare.None, bufferSize: 64, useAsync: true))
    {
fs.Write(encoding, 0, encoding.Length);
    }
}
```

Once we run the application now we will see the following exception which is expected because the file is locked by one of the Thread for adding data and another thread parallelly tries to do the same thing and raises an access exception:

System.IO.IOException

HResult=0x80070020

Message=The process cannot access the file ‘..\\netcoreapp3.1\\File0.txt’ because it is being used by another process.

In this scenario, we can use the lock as well and will get the same output; however, it doesn't make sense to lock writing into a different file as the lock will allow to writing into any file sequentially, that is, if currently, a thread is writing into file1, the lock will block writing into any other file also, and that's why a named mutex would be better here, considering the performance impact as code is blocked only for specific files. For testing purpose removing mutex and adding a lock would result in a significant dip in performance which we can see in the following output:

```
Writing file to disk
Time ellapsed 3028
Writing file to disk completed
```

Figure 7.5: Output of FileUpload application with synchronization using lock

So with this, we can say lock and Mutex can be used to achieve synchronization; however, to lock a block of code or a resource across process named Mutex can be used.

Some essential facts about Mutex:

- A mutex has thread affinity, so the thread locking a resource needs to unlock the resource; that is, locking and unlocking has to happen on the same thread.
- A mutex can spawn across the process.
- A lock is acquired by calling the `WaitOne` method and released using the `ReleaseMutex` method; however, `WaitOne` can be called multiple times on

the same thread, which is also known as a **recursive mutex**; however, we need to ensure that `ReleaseMutex` is called as many times as `WaitOne` is called.

In the preceding example, we can use `WriteAsync` instead of `Write` however that will result in an exception as `Mutex` has thread affinity which means thread calling `WaitOne` needs to call release method, and since the code after `await` would run on a different thread, it would give an exception – “*Object synchronization method was called from an unsynchronized block of code. Exception on Mutex.Release()*” (<https://stackoverflow.com/questions/9017521/object-synchronization-method-was-called-from-an-unsynchronized-block-of-code-e>).

To avoid this exception, we need to use an advanced synchronization construct called `AutoResetEvent`, which we will see later in this chapter.

SpinLock (Exclusive)

`Spinlock` is another form of exclusive lock which will synchronize access to the shared resource; however, there isn’t thread context switching. So going back to all other locking techniques whenever a thread is blocked to access a shared resource, it stops consuming any CPU cycles by giving up its processor time slice and causing a context switching in a thread, the same thing happens when the thread is unblocked from the blocked state. Although this context switching leads only to a few milliseconds delay at a large scale, this is still overhead.

So if there is a shared resource and needs locking for a very few milliseconds, it would be better not to block all the threads that need to access shared resources, but just to continue spinning, which is something like calling a while loop until the shared resource is unblocked. It can be achieved in .NET using a `SpinLock` class; let’s take the `AddBalanceToAccount` method of `BankAccount` class this time we will use a `SpinLock` to synchronize access to `accountBalance` variable. We will first declare an object of `SpinLock` class that looks like this:

```
SpinLock spinLock = new SpinLock();
```

Then use this lock to protect variables `accountBalance` and `numberOfTransactions` by calling `Enter` method of `SpinLock` class, this method accepts a Boolean variable which needs to be `false` before calling this method as once a lock is acquired `spinlock` set this variable to `true`. This variable helps if there is an exception after the lock is acquired, lock can be released safely. It is how our `AddBalanceToAccount` method will look like:

```
async Task AddBalanceToAcccount(long amount)
{
    await Task.Delay(1);
```

```
    bool lockAcquired = false;
    try
    {
        spinLock.Enter(ref lockAcquired);
        accountBalance = accountBalance + amount;
        numberOfTransactions = numberOfTransactions + 1;
    }
    finally
    {
        if (lockAcquired)
        {
            spinLock.Exit();
        }
    }
}
```

Modify `AddMoneyToAccountAsync` to include a `Timer`, it's code will look like below:

```
public async Task AddMoneyToAccountAsync()
{
    Stopwatch timer = new Stopwatch();
    timer.Start();

    var tasks = new Task[99999];
    for (int i = 1; i <= tasks.Length; i++)
    {
        tasks[i - 1] = AddBalanceToAcccount(i);
    }

    await Task.WhenAll(tasks);

    Console.WriteLine($"Time taken - {timer.ElapsedMilliseconds}");
}
```

Once we run `AddMoneyToAccountAsync` output will look like the following screenshot:

```
Initial Balance 0
Time taken - 12575
Current Balance 4999950000, total number of transactions - 99999
```

Figure 7.6: Output of using a spinlock

Just for testing purposes replacing `spinlock` with a lock and checking output, we get the output as shown in *Figure 7.6*. Here we can see that time taken slightly goes up, and that's because there isn't any context switching with `spinlock`:

```
Initial Balance 0
Time taken - 13779
Current Balance 4999950000, total number of transactions - 99999
```

Figure 7.7: Output of using lock

Some essential facts about `SpinLock`:

- Use `spinlock` only for locking code that executes fast perhaps few microseconds, that is, a brief spinning is preferred over-blocking
- Always prefer locks over `spinlock` as `spinlock` although consumes fewer times in some scenarios consume a lot of CPU

In this section, we have seen what `spinlock` is, why we need it, and how we can implement it. In general, `SpinLock` is more of a subjective concept that is used as part of some of the .NET classes especially slim versions of synchronization constructs.

Semaphore (Non-Exclusive)

`Semaphore` is a non-exclusive lock that supports synchronization by limiting access to a limited number of threads. So unlike `mutex`, which allows only one thread to enter critical section `semaphore` allows a set of threads to enter the critical section, the number of threads that have access to shared resources is defined, which creating `semaphore`. `Semaphore` is a non-exclusive lock, and hence it should be used in situations where we need to lock a pool of resources for example in a client-server scenario (where you own both client app and server APIs) say you want to restrict several calls your client app can make to your API concurrently from within a single instance or something like thread pool, or database connection `semaphores` are an ideal fit.

In .NET `semaphores` can be created using `System.Threading.Semaphore` class (<https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphore?view=netcore-3.1>), an object of `Semaphore` class needs to be instantiated which has multiple constructors, but two essential parameters need to be passed always

1. The initial number of entries
2. Maximum number of concurrent entries

So, a typical initialization will look like the following code:

```
Semaphore semaphore = new Semaphore(0, 3);
```

In this case, we are telling initial concurrent requests allowed is 0 and will allow upto 3 concurrent threads after release; that is, in this case, we are initializing semaphore; however, the program has to wait until a release is called at least once. If semaphore needs to enter semaphore immediately and allow maximum concurrency ideally both the parameters should have the same value, something like the following code:

```
Semaphore semaphore = new Semaphore(3, 3);
```

To acquire a semaphore, we need to call the `WaitOne` method of `semaphore` class and to release the need to call the `Release` method, which accepts an optional integer to release semaphore that many numbers of times, if nothing is passed semaphore releases one thread. We will see this with a simple example of building water where water needs two Hydrogen threads and one Oxygen thread in this sequence. The restriction is that all the threads from one molecule bond before subsequent molecules from any other thread. We will take an input of a series of strings a combination of 'H' and 'O' and handle accordingly. So, if the input is HHHHOO output would be HHOHHO, that is, each character is processed by a thread, and after the second H, the program needs to wait for next O in the sequence to complete water molecule before processing other characters in sequence.

We will start with first creating Water class, and we need two semaphores, one for Hydrogen and another for Oxygen. Now since for every water molecule two Hydrogen threads are required we will initialize Hydrogen semaphore with a maximum concurrent thread count of 2, and since Hydrogen thread can be processed as soon as it is created the first number of entries can be 0, so this will look like the following code:

```
Semaphore semaphoreH = new Semaphore(0, 2);
```

Then we will create Oxygen semaphore, which will have a maximum concurrent thread count of 1, and since Oxygen thread always needs to be processed/released only after two Hydrogen threads, initial concurrent requests should be 0 that is Oxygen thread needs to wait till any two Hydrogen threads are processed. With this Oxygen semaphore will look like the following code:

```
Semaphore semaphoreO = new Semaphore(0, 1);
```

So, our class will look like the following code block:

```
public class Water
```

```

{
    Semaphore semaphore0 = new Semaphore(0, 1);
    Semaphore semaphoreH = new Semaphore(2, 2);
}

```

Now add two private methods to print Hydrogen and Oxygen like below:

```

void ReleaseHydrogen()
{
    Console.WriteLine("H");
}

void ReleaseOxygen()
{
    Console.WriteLine("O");
}

```

Now we need two more methods to process Hydrogen and Oxygen threads and release each other accordingly; this is where we will use our semaphores:

- Hydrogen method will allow entering two threads in the critical section and will release Oxygen semaphore when two Hydrogen threads are processed, but won't allow more than two Hydrogen threads to enter critical section until one Oxygen thread is processed
- Oxygen method will wait to release Oxygen if two Hydrogen threads are processed and release Hydrogen semaphore twice or else will wait on Oxygen semaphore

The following code will show both the methods:

```

int hCount = 0;

public async Task HThread(Action releaseH)
{
    await Task.Delay(1);

//Wait on Hydrogen thread, code after this will be blocked after
processing two Hydrogen threads until one Oxygen thread is processed
semaphoreH.WaitOne();

releaseH();

```

```
hCount++;

    if (hCount % 2 == 0) //For every two Hydrogen threads
releasing Oxygen semaphore to process Oxygen method.

    {

semaphore0.Release();

    }

}

public async Task OThread(Action releaseO)
{
    await Task.Delay(1);

//Locking on Oxygen semaphore, this will allow being processed only when
2 Hydrogen threads are processed or else will wait.

//Code after this is blocked until two Hydrogen threads are processed as
initial concurrent threads for Oxygen semaphore is 0 (first parameter)

semaphore0.WaitOne();

releaseO();

semaphoreH.Release(2); //Exiting Hydrogen semaphore twice, allowing two
Hydrogen to be processed
}
```

Now let's define a method to build water in which we will loop through input sequence and initiate Hydrogen thread if the input character is H and initiate Oxygen thread if the input character is O:

```
public async Task BuildWaterAsync(string input)
{
    List<Task> tasks = new List<Task>();
    foreach (char c in input)
    {

        switch (c)
        {

            case '0':
```

```
tasks.Add(OThread(ReleaseOxygen));
        break;
    case 'H':
tasks.Add(HThread(ReleaseHydrogen));
        break;
    default:
        break;
    }
}
await Task.WhenAll(tasks);
}
```

Let's consume this class in a simple console application and call `BuildWaterAsync` method, so create a console application add `Water` class to the app and replace the main method with the following code:

```
static async Task Main(string[] args)
{
    Water water = new Water();
    while (true)
    {
        Console.WriteLine("Please enter sequence of Hydrogen and
Oxygen molecules or e to Exit");
        string input = Console.ReadLine();
        if (input == "e")
            break;
        await water.BuildWaterAsync(input);
    }
}
```

Once we run this code output will look below:

```
Please enter sequence of Hydrogen and Oxygen molecules or e to Exit
OOHHHH
H
H
O
H
H
O
Please enter sequence of Hydrogen and Oxygen molecules or e to Exit
HHHHOO
H
H
O
H
H
O
Please enter sequence of Hydrogen and Oxygen molecules or e to Exit
HOHOHH
H
H
O
H
H
O
Please enter sequence of Hydrogen and Oxygen molecules or e to Exit
```

Figure 7.8: Output of build water application using semaphore

In this sample, we can see how semaphore is helping to lock the critical section with more than one thread and signaling on the availability of resources. Similarly, many classic synchronization problems like the Dining Philosopher problem, producer-consumer problem, the reader-writer problem can be solved using semaphores.

Note: The above example is built with the assumption that the user will always enter a combination that can be converted into one water molecule as the intent is here is to understand semaphore. Further validation can be added to handle scenarios where there aren't enough Hydrogen threads or Oxygen threads.

Some essential facts about semaphore:

- Semaphore doesn't have thread affinity, so any thread can call Release method, its application responsibility to release semaphore appropriately
- Semaphores are usually used for signaling of resource availability, like a thread is available in the thread pool
- Semaphores can be named semaphores; these can be used to support across process synchronization

- A typical scenario of semaphore usage is a requirement where we want to limit concurrent database connections or in a multi-core scenario to limit the number of concurrent threads executing a specific operation.

In this section, we have seen what semaphore is, why we need it, and how to implement it. In the next section, we will see a lightweight version of Semaphore.

SemaphoreSlim (Non-exclusive)

`SemaphoreSlim` is another class in `System.Threading` and is a lightweight version to create Semaphores in C#. When we use the `System.Threading.Semaphore` class to create a semaphore it internally uses Windows kernel semaphores which involves blocking, context switching of threads, and also expensive kernel transition; however, `SemaphoreSlim` implements spinning through `SpinWait`, and if it cannot acquire lock after spinning for a while (microseconds), then it uses blocking to acquire the lock. As discussed earlier that spinning for a very brief period is less expensive as compared to blocking; hence `SemaphoreSlim` is a good fit for such scenarios where the wait time to acquire critical section is less. Couple of other properties that `SemaphoreSlim` supports are:

- To acquire a slot in `SemaphoreSlim`, we need to call the `Wait` method or `WaitAsync` method.
- `SemaphoreSlim` doesn't support named semaphores, so by default, it's always local semaphore.
- `SemaphoreSlim` has support for `async` methods like `WaitAsync`.
- Since `async` methods are available, `SemaphoreSlim` also allows cancellation token, which means cancellation is allowed and, at times, can be useful to come out of the deadlock.
- `SemaphoreSlim` has a constructor that supports initializing it with one parameter, which is the first available slots and no upper limit. In such a semaphores release method can be called any number of times, and there won't be any exception (`SemaphoreFullException`) thrown in such cases, it's developer's responsibility to call wait and release methods appropriately.
- `SemaphoreSlim` has `CurrentCount` property, which tells the number of threads that can get a slot.
- Just like semaphore, `SemaphoreSlim` is also threaded agnostic.

Going back to our example if we use `SemaphoreSlim` we need to change initialization as below:

```
SemaphoreSlim semaphoreH = new SemaphoreSlim(2, 2);
SemaphoreSlim semaphoreO = new SemaphoreSlim(0, 1);
```

Instead of `WaitOne`, we will call `WaitAsync` and making use of `CurrentCount` our code will look like the following:

```
public async Task HThread(Action releaseH)
{
    if (semaphoreH.CurrentCount == 0 && semaphoreO.CurrentCount
== 1)
    {
        Console.WriteLine("Hydrogen is ready, waiting for
Oxygen");
    }
    //Wait on Hydrogen thread, code after this will be blocked
    //after processing two Hydrogen threads until one Oxygen thread is
    //processed
    await semaphoreH.WaitAsync();
    releaseH();
    hCount++;
    if (hCount % 2 == 0) //For every two Hydrogen threads
    //releasing Oxygen semaphore to process Oxygen method.
    {
        semaphoreO.Release();
    }
}
public async Task OThread(Action releaseO)
{
    if (semaphoreH.CurrentCount > 0 && semaphoreO.CurrentCount
== 0)
    {
        Console.WriteLine("Oxygen is ready, waiting for
Hydrogen");
    }
    //Locking on Oxygen semaphore, this will allow being
    //processed only when 2 Hydrogen threads are processed or else will wait.
    //Code after this is blocked until two Hydrogen threads are
    //processed as initial concurrent threads for Oxygen semaphore is 0 (first
    //parameter)
    await semaphoreO.WaitAsync();
```

```

        release0();

        semaphoreH.Release(2); //Exiting Hydrogen semaphore twice,
allowing two Hydrogen to be processed
    }
}

```

Output for with `SemaphoreSlim` will have additional information as we now can tell when Hydrogen is waiting on Oxygen or vice-versa. Once we run this application output will be as shown in *Figure 7.9*:

```

Please enter sequence of Hydrogen and Oxygen molecules or e to Exit
OOHHHHHHHHOO
Oxygen is ready, waiting for Hydrogen
Oxygen is ready, waiting for Hydrogen
H
H
O
H
H
O
H
H
Hydrogen is ready, waiting for Oxygen
Hydrogen is ready, waiting for Oxygen
O
H
H
O

```

Figure 7.9: Output of build water application using SemaphoreSlim

As such, there is no hard and fast rule on what to use when but mostly by the rule of elimination, where if we want cross-process semaphore to use `Semaphore` class, if we want lightweight semaphore for synchronization of a resource that is held for a very shorter period go for `SemaphoreSlim`.

Reader/Writer locks (Non-Exclusive)

Often, it's a case where we use exclusive locks for a shared resource, which is recommended practice and also guarantees proper synchronization of data; however, there could be scenarios where a resource just read multiple times with periodic updates. Using lock-in such scenarios will ensure synchronization during concurrent access; however, it will slow down the application as two threads that just wanted to read data will be processed sequentially. Such cases can be better handled using `ReaderWriter` locks, which allows a shared resource to be accessed by multiple threads that wanted to perform read operation and allows a single thread for the write operation.

Taking an analogy of a teacher writing on blackboard and students copying it:

- While the teacher is writing on the blackboard, none of the students can see what is on the blackboard (Single write)
- The teacher won't erase content on blackboard until the last student finishes copying the content (Multiple reads and write a thread in queue until last read thread has released the lock)

A `ReaderWriter` lock can be achieved in C# either by creating an object of `System.Threading.ReaderWriterLock` or `System.Threading.ReaderWriterLockSlim` class. `ReaderWriterLockSlim` is a thinner version of `ReaderWriterLock`, which lesser memory footprint and better performing. Both of these classes have the following methods:

- To acquire read lock which can be called by multiple threads—`AcquireReaderLock/EnterReadLock`
- To release read lock acquired by a thread (`ReadWriteLock` have thread affinity so same thread that acquired lock needs to release the lock, this is applicable for both read and write locks)—`ReleaseReaderLock/ExitReadLock`
- To acquire write lock—`AcquireWriterLock/EnterWriteLock`
- To release write lock—`ReleaseWriteLock/ExitWriteLock`
- To acquire an upgradeable lock. An upgradeable lock helps in acquiring a read lock and then upgrade to write lock based on a condition, for example, an upset scenario. It can be normally achieved by acquiring a read lock, check if data is present, if not release read lock and then acquire a write lock. However, the state of the shared resource may not remain the same between releasing read lock and acquiring write lock; hence it is preferable to use an upgradeable lock.

Let's see this with a simple example of writing data into the file, where we simulate around 30 threads with multiple reads and periodic updates. Let's create a console application and create a new public class `FileWrite`, create an instance of `ReaderWriterLockSlim` and also add a timer that will be used to get the execution time, our class will look like below:

```
public class FileWrite
{
    Stopwatch timer; //To compare performance with Monitor
    public FileWrite()
    {
        timer = new Stopwatch();
        timer.Start();
    }
}
```

```

const string fileName = "SampleReadLock.txt";
    ReaderWriterLockSlim readerWriterLockSlim = new
ReaderWriterLockSlim();
}

```

Add two methods.

- **private void ReadFile():** A method that reads data from the file, we will lock the read operation using `EnterReadLock` and `ExitReadLock` method of `ReaderWriterLock` to avoid trying to read the file when it is open for writing data (if we do not lock the read operation `FileStream` will throw `System.IO.IOException`—the process cannot access the file). This method will look like the following code:

```

private void ReadFile()
{
    if (File.Exists(fileName))
    {
        readerWriterLockSlim.EnterReadLock();
        using (FileStream fs = new FileStream(fileName,
 FileMode.Open, FileAccess.Read, FileShare.Read, 2048, useAsync:
 true))
        {
            using (System.IO.StreamReader rdr = new
System.IO.StreamReader(fs))
            {
                Thread.Sleep(500); //Used to perform
timer calculation,
                Console.WriteLine(rdr.ReadToEnd());
            }
        }
        readerWriterLockSlim.ExitReadLock();
    }
}

```

- **private void WriteFile(int lineNumber):** A method that writes into file but before writing into file acquires a `ReaderWriterLock` using `EnterWriteLock` and `ExitWriteLock`. This method implementation will look like the following code:

```
private void WriteFile(int lineNumber)
```

```
        {
            readerWriterLockSlim.EnterWriteLock();
            string text = $"Line {lineNumber} ReadWriteLock" +
Environment.NewLine;
            byte[] encoding = Encoding.ASCII.GetBytes(text);
            using (FileStream fs = new FileStream(fileName,
 FileMode.Append, FileAccess.Write, FileShare.Write, 2048,
useAsync: true))
            {
                fs.Write(encoding, 0, encoding.Length);
            }
            readerWriterLockSlim.ExitWriteLock();
        }
    }
```

Now add a method that will write into the file if a text is present else display, so primarily an upsert operation. This method will make use of `EnterUpgradeableReadLock`, which can be used to upgrade to write lock conditionally. This method of implementation will look at the following code:

```
private void ReadorUpdateFile()
{
    string fileContent = String.Empty;
    if (File.Exists(fileName))
    {
        readerWriterLockSlim.EnterUpgradeableReadLock();
        //First read the contents and if specific content exists
        then print on console else write into file
        using (FileStream fs = new FileStream(fileName, FileMode.
Open, FileAccess.Read, FileShare.Read, 2048, useAsync: true))
        {
            using (System.IO.StreamReader rdr = new System.
IO.StreamReader(fs))
            {
                fileContent = rdr.ReadToEnd();
            }
        }
        if (!(fileContent.Contains("Line 15")))
        {
```

```

        readerWriterLockSlim.EnterWriteLock();

        using (FileStream fswrite = new FileStream(fileName,
FileMode.Append, FileAccess.Write, FileShare.Write, 2048, useAsync:
true))
{
    byte[] encoding = Encoding.ASCII.GetBytes($"Line
15 ReadWriteLock" + Environment.NewLine);
    fswrite.Write(encoding, 0, encoding.Length);
}
readerWriterLockSlim.ExitWriteLock();
}

else
{
    Thread.Sleep(500); //Used to perform timer
calculation,
    Console.WriteLine(fileContent);
}
readerWriterLockSlim.ExitUpgradeableReadLock();
}
}

```

Let's add another method to call these methods; let's call it `PerformFileOperation`. The purpose of this method is to simulate around 30 parallel requests and primarily calling `ReadFile` and conditionally calls `WriteFile`, `ReadorUpdateFile` methods:

```

public async Task PerformFileOperation()
{
    var tasks = new Task[31];
    for (int i = 0; i < tasks.Length; i++)
    {
        if (i % 10 == 0) //Calling write every tenth time
        {
            tasks[i] = Task.Run(() => WriteFile(i + 1));
            Thread.Sleep(1000); //Used to perform timer calculation
        }
        else if (i == 15 || i == 21) //Calling upsert twice
        {

```

```
        tasks[i] = Task.Run(() => ReaderUpdateFile());
    }
    else //Calling read most of the time
    {
        tasks[i] = Task.Run(() => ReadFile());
    }
}
await Task.WhenAll(tasks);
Console.WriteLine($"Time elapsed {timer.ElapsedMilliseconds}"); //Displaying time taken for execution
readerWriterLockSlim.Dispose();
}
```

Let's instantiate this class in our `Main` method and call the `PerformFileOperation` method to read from a file and write/update it into the same file conditionally. The `Main` method will look at the following code:

```
static async Task Main(string[] args)
{
    Console.WriteLine("Writing file to disk");
    FileWritefileupload = new FileWrite();
    await fileupload.PerformFileOperation();
    Console.WriteLine("Writing file to disk completed");
    Console.Read();
}
```

Once we run this application, the output is shown in the following screenshot:

```

Line 1 ReadWriteLock
Line 11 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Line 1 ReadWriteLock
Line 11 ReadWriteLock
Line 15 ReadWriteLock
Line 21 ReadWriteLock

Time elapsed 4119
Writing file to disk completed

```

Figure 7.10: Output of file read application using ReaderWriteLockSlim

Now let's remove `ReadWriteLock` and replace the synchronization mechanism with `lock` and run the application; we noticed that output remains the same; however,

execution time is much higher (almost three times), and we know that is expected because lock won't allow multiple threads concurrently. So, `ReaderWriteLock` is an effective synchronization mechanism for a shared resource when there are many reads but periodic updates.

Some essential facts about reader/write locks:

- `ReaderWriteLock` allows multiple reads, one exclusive write lock
- Among multiple reads, one can be upgraded to write lock
- `ReaderWriteLock` has thread affinity

In this section, we have seen what `ReaderWriteLock` is, why we need it, and how to implement it. In the next section, we will see what thread signaling, how to implement it, and specific scenarios where thread signaling is helpful is.

Signaling constructs

Signaling constructs are the synchronization primitives available in C# that help in signaling a thread to wait or proceed based on a notification. A simple example would be `Thread.Join` where say thread X calling the `join` method on thread Y will wait until thread Y is completed. Other signaling constructs that are available in C# are `AutoResetEvent`, `ManualResetEvent/ManualResetEventSlim`, `CountdownEvent` and `Barrier` class. C# provides an `EventWaitHandle` class that is used for thread synchronization and `AutoResetEvent`, `ManualResetEvent` implements this class. A simple comparison of these constructs is as following:

Signaling construct	Usage
<code>AutoResetEvent</code>	Allows unblocking a thread (or a thread once) through signaling.
<code>ManualResetEvent/ManualResetEventSlim</code>	Allows unblocking all the threads (or a thread indefinitely) through signaling and blocks only after manually resetting signaling status through the <code>Reset</code> method.
<code>CountdownEvent</code>	Allows unblocking a thread after it receives a predefined number of signals.
<code>Barrier</code>	The barrier allows threads to execute a piece of code parallel and wait till all the threads have completed the execution of that code with a post-execution phase.

Table 7.1: Comparison of Signaling constructs

Let us deep-dive into each of these constructs in the next section.

AutoResetEvent

`AutoResetEvent` is like a toll gate where only one car is allowed at a time, and each car needs access to go through the gate. The `AutoResetEvent` class helps in creating a signaling construct that allows sending a signal to unblock a blocked/waiting thread and immediately reset the signals, which means any subsequent thread will continue to wait in a queue until next signal is received.

An `AutoResetEvent` class is instantiated by its constructor something like this:

```
AutoResetEvent event_1 = new AutoResetEvent(false);
```

The parameter to this constructor signifies if the event is already signaled or non-signaled, that is true to signal, which means one thread can proceed to process(tollgate is by default open for one car) and the parameter is passed as the false thread will wait to receive a signal. Blocking a thread is done by calling `WaitOne()`, and releasing/signaling one thread is done by calling the `Set()` method. Let's look at this with a simple example of stock trading where stock needs to be purchased when it reaches a specific price; we will use two threads here:

- One to take input from the user on the price that stock needs to be purchased
- One to randomize the current stocks price and see if it matches with user input and place order

Once the user inputs stock buy price, to complete the order, that thread (waiting thread) will continue to wait till there is a match in the current stock price. Now since matching is a complex process, we will run on a separate thread. Once there is a match, a signal (signaling thread) is sent to complete the order. We will use `AutoResetEvent` to handle this signaling across threads. Let's create a class called `StockTrading` as following:

```
public class StockTrading
{
    AutoResetEvent autoResetEvent = new AutoResetEvent(false);

    public int currentStockPriceOfXYZ{ get; set; } //Holds current
    stock price

    public int buyPriceofXYZ{ get; set; } //Buy price of stocks

    public bool StockPurchased{ get; set; } //Flag that is set to
    true once order is successful

    public StockTrading(bool stockPurchased)
    {
        this.StockPurchased = stockPurchased;
    }
}
```

```
    }  
}
```

Add the following two methods:

- `PlaceOrder()`: One to place an order and will wait for the signal from the following method to complete the order
- `ValidatePrice()`: One to validate stock price with the buy price and signal accordingly. This method is called every time there is a change in stock price:

```
public void PlaceOrder()  
{  
    Console.WriteLine("Enter price at which you want to  
buy XYZ (minimum 1, maximum 5)");  
    buyPriceofXYZ = Convert.ToInt32(Console.ReadLine());  
    this.StockPurchased = false;  
    autoResetEvent.WaitOne(); //Wait until receives signal from price  
validation  
    Console.WriteLine($"Stock purchased at buy price of  
{buyPriceofXYZ}");  
    Console.WriteLine("One stock order is completed;  
press enter to exit");  
    this.StockPurchased = true;  
    Console.ReadLine();  
}  
  
public void ValidatePrice()  
{  
    if (this.buyPriceofXYZ == this.  
currentStockPriceOfXYZ)  
    {  
        Console.WriteLine($"Current stock price of {this.  
currentStockPriceOfXYZ} is matching with buy price of {this.  
buyPriceofXYZ}");  
        autoResetEvent.Set(); //Signal first thread waiting in queue to  
execute  
    }  
    else if(!this.StockPurchased)
```

```
{  
    Console.WriteLine($"Current stock price of {this.  
currentStockPriceOfXYZ} is not matching with buy price of {this.  
buyPriceofXYZ}");  
}  
}
```

Let's add our class to a console application and consume it in our main application as following code:

```
static void Main(string[] args)  
{  
    Console.WriteLine("Please enter buy price of stock XYZ");  
    StockTrading stockTrading = new StockTrading(false);  
  
    //Thread to place order  
    Thread placeOrder = new Thread(stockTrading.PlaceOrder);  
    placeOrder.Start();  
  
    //Thread that checks for current price and completes order  
    Thread validatePrice = new Thread(() =>  
    {  
        Random randomCurrentPriceofStock = new Random();  
        while (!stockTrading.StockPurchased)  
        {  
            stockTrading.currentStockPriceOfXYZ =  
randomCurrentPriceofStock.Next(1, 5);  
            stockTrading.ValidatePrice();  
            Thread.Sleep(1000); // Wait for input before  
executing next iteration or else screen will overflow  
        }  
    });  
    validatePrice.Start();  
}
```

Here we are creating two threads to place an order and validate the price; validate price thread needs to be emphasized as stock prices change. Once we run this application output will look like the following screenshot:

```
Please enter buy price of stock XYZ
Enter price at which you want to buy XYZ (minimum 1, maximum 5)
Current stock price of 3 is not matching with buy price of 0
3
Current stock price of 2 is not matching with buy price of 3
Current stock price of 2 is not matching with buy price of 3
Current stock price of 4 is not matching with buy price of 3
Current stock price of 4 is not matching with buy price of 3
Current stock price of 1 is not matching with buy price of 3
Current stock price of 4 is not matching with buy price of 3
Current stock price of 2 is not matching with buy price of 3
Current stock price of 1 is not matching with buy price of 3
Current stock price of 2 is not matching with buy price of 3
Current stock price of 4 is not matching with buy price of 3
Current stock price of 2 is not matching with buy price of 3
Current stock price of 3 is matching with buy price of 3
Stock purchased at buy price of 3
One stock order is completed, press enter to exit
```

Figure 7.11: Output of stock trading application using AutoResetEvent

As you can see, due to the calling of `WaitOne`, `PlaceOrder()` will wait until `Set()` is called; in our case, we are calling it once purchase price matches with the stock price.

Some important facts about the `AutoResetEvent` class:

- Calling `Set` multiple times will not cause an exception even if there aren't any waiting threads; all it does is not block the first thread that is calling `WaitOne`, but subsequent threads will be blocked until another thread is calling `Set`. For example, in the below-console app:

```
private static AutoResetEvent autoResetEvent = new
AutoResetEvent(false);

static void Main()
{
    for (int x = 0; x < 3; x++)
    {
        Thread thread = new Thread(ThreadProcecss);
        thread.Name = "Thread " + x;
        thread.Start();
    }
}
```

```
        }

        Console.WriteLine("Press Enter to release blocked
threads");

        Console.ReadLine();

autoResetEvent.Set(); //Thread 0 is released
autoResetEvent.Set(); //Thread 1 is released
autoResetEvent.Set(); //Thread 2 is released
autoResetEvent.Set(); //Thread 3 won't be blocked
autoResetEvent.Set(); // This is of no use as any set call after
above line will be nullified once a thread goes through
autoResetEvent.Set(); // This is of no use as any set call after
above line will be nullified once a thread goes through

for (int x = 3; x < 7; x++)
{
    Thread thread = new Thread(ThreadProcecss);
    thread.Name = "Thread " + x;
    thread.Start();
}

static void ThreadProcecss()
{
    string threadName = Thread.CurrentThread.Name;
    Console.WriteLine($"{threadName} waits on AutoResetEvent.");
    autoResetEvent.WaitOne();
    Console.WriteLine($"{threadName} is released from
AutoResetEvent...");
}
```

Once we run this code, the output will look like the following screenshot:

```
Thread 0 waits on AutoResetEvent.
Thread 1 waits on AutoResetEvent.
Thread 2 waits on AutoResetEvent.
Press Enter to release blocked threads

Thread 0 is released from AutoResetEvent..
Thread 1 is released from AutoResetEvent..
Thread 2 is released from AutoResetEvent..
Thread 3 waits on AutoResetEvent.
Thread 3 is released from AutoResetEvent..
Thread 4 waits on AutoResetEvent.
Thread 5 waits on AutoResetEvent.
Thread 6 waits on AutoResetEvent.
```

Figure 7.12: Output of console application using AutoResetEvent with multiple calls to Set()

- As `AutoResetEvent` class inherits from `EventHandle` it can also be created like the following code:

```
EventWaitHandle autoResetEvent = new EventWaitHandle(false,
EventResetMode.AutoReset);
```
- `AutoResetEvent` is thread agnostic as it is more of signaling construct

In this section, we have seen what `AutoResetEvent` is, why we need it, and how to implement it. In the next section, we will see another signaling construct known as `ManualResetEvent`, which is very similar to `AutoResetEvent`.

ManualResetEvent/ManualResetEventSlim

`ManualResetEvent` is another signaling construct like `AutoResetEvent`, which can be used for by threads to signal a different thread. The difference between `AutoResetEvent` and `ManualResetEvent` is that it can unblock all the blocked threads until it is manually reset. Taking an analogy `ManualResetEvent` is just like a gate, which is when opened allows all the people waiting outside to come in until it is closed.

A printer can also be an excellent example of `ManualResetEvent` that is initialized in the signaled state where all the print jobs that come to a printer are executed immediately until you run out of paper or ink, and that's where all the `ManualResetEvent` is non-signaled, and all the printing jobs are paused. The moment we fill the ink or add paper, `ManualResetEvent` is signaled, and all the printing jobs are continued until something again blocks them.

`ManualResetEvent` has three methods to achieve signaling:

- **Set:** The `Set` method is called by one thread to send the signal to all the waiting threads. Unlike `AutoResetEvent`, the signal is received by all waiting threads.

- **WaitOne/Wait:** Any thread that calls `WaitOne/Wait` is blocked until it is signaled, if a thread has already received signaled then none of the threads would be blocked.
- **Reset:** This method is used to reset `ManualResetEvent` to a non-signaled state. If a call to `Set` is not followed by a call to `Reset`, all the threads calling `WaitOne` aren't blocked until `Reset` is called.

`ManualResetEvent` can be constructed by calling the `ManualResetEvent` constructor which accepts a Boolean value like the following code:

```
ManualResetEvent manualResetEvent = new ManualResetEvent(false);
ManualResetEventSlim manualResetEvent = new ManualResetEventSlim(false);
```

The Boolean flag has the same impact as in `AutoResetEvent`; that is, if initialized with `true`, it won't block any code by default until `ManualResetEvent` is reset (call `Reset`) and `false` means threads will be blocked as soon as they see a call to `WaitOne` method of `ManualResetEvent`. The `ManualResetEvent` class can also be constructed through `EventWaitHandle`:

```
EventWaitHandle manualResetEvent = new EventWaitHandle(false,
EventResetMode.ManualReset);
```

`ManualResetEvent` also has a lightweight class `ManualResetEventSlim`, this one doesn't use operating system objects (kernel objects) directly and uses spinning for a shorter period before blocking and then finally fall back to kernel objects hence is much faster and lighter than `ManualResetEvent`. Let's use the stock example we have used for `AutoResetEvent`, and this time, say we have to allow multiple threads to successfully place an order if there is a specific match, so it can be one or more than one threads that can match a particular stock order. Let's start by modifying `StockTrading` class by adding an object of `ManualResetEventSlim`:

```
public ManualResetEventSlim manualResetEvent = new
ManualResetEventSlim(false);
```

Now modify `PlaceOrder()` to call `Wait` instead of `WaitOne` so that our method looks like the following code:

```
public void PlaceOrder(int threadId, int buyPrice)
{
    buyPriceofXYZ = buyPrice;
    this.StockPurchased = false;
    manualResetEvent.Wait(); //Wait until receives signal from price validation
    Console.WriteLine($"Stock purchased at buy price of {buyPriceofXYZ}, Stock order {threadId} is completed");
```

```
this.StockPurchased = true;
    }
```

Update ValidatePrice() method with ManualResetEvent variable, so our method looks like below:

```
public void ValidatePrice()
{
    if (this.buyPriceofXYZ == this.currentStockPriceOfXYZ)
    {
        Console.WriteLine($"Current stock price of {this.
currentStockPriceOfXYZ} is matching with buy price of {this.
buyPriceofXYZ}");
    manualResetEvent.Set(); //Signal first thread waiting in queue to execute
    }
    else if(!this.StockPurchased)
    {
        Console.WriteLine($"Current stock price of {this.
currentStockPriceOfXYZ} is not matching with buy price of {this.
buyPriceofXYZ}");
    }
}
```

Now while consuming, we will create multiple place orders with the same buy price, and the expectation is that all the orders are placed. So, create a console application and add StockTrading class; in the main method, add logic to create multiple threads for place order and then another thread calling ValidatePrice to complete order. With this, our Main method will look like the following code:

```
static void Main(string[] args)
{
    Console.WriteLine("Please enter buy price of stock XYZ");
    StockTrading stockTrading = new StockTrading(false);
    Console.WriteLine("Enter price at which you want to buy XYZ
(minimum 1, maximum 5)");
    int buyPrice = Convert.ToInt32(Console.ReadLine());
    //Multiple threads to place 3 orders
    for (int i = 0; i < 3; i++)
    {
```

```

        Thread placeOrder = new Thread(() => stockTrading.
PlaceOrder(i, buyPrice));
        Thread.Sleep(1000);
        placeOrder.Start();
    }

    Console.WriteLine("3 orders placed; press enter to start
stock price matching!!");
    Console.ReadLine();
    //Thread that checks for current price and completes order
    Thread validatePrice = new Thread(() =>
{
    Random randomCurrentPriceofStock = new Random();
    while (!stockTrading.StockPurchased)
    {
        stockTrading.currentStockPriceOfXYZ =
randomCurrentPriceofStock.Next(1, 5);
        stockTrading.ValidatePrice();
        Thread.Sleep(1000); // Wait for input before
executing next iteration or else screen will overflow
    }
});
validatePrice.Start();
Console.ReadLine();
}

```

If we run code now we will see that all the orders successfully placed once there is a price matching in `ValidatePrice()` method, however let's create some more orders after initial set of orders and call `ValidatePrice` method to complete the order. So, add the following code to main method after first set of orders:

```

//Resetting ManualResetEvent to non-signaled state so that
any subsequent orders are blocked (threads)
//if this is not called call to .Wait method won't be
blocked (Gate is open till Reset is called)
stockTrading.manualResetEvent.Reset();
Console.WriteLine("\nPlease enter buy price of stock XYZ");
buyPrice = Convert.ToInt32(Console.ReadLine());

```

```
//Multiple thread to place 2 more orders
for (int i = 3; i < 5; i++)
{
    Thread placeOrder = new Thread(() => stockTrading.
PlaceOrder(i, buyPrice));
    Thread.Sleep(1000);
    placeOrder.Start();
}
Console.WriteLine("2 orders placed; press enter to start
stock price matching!!");
Console.ReadLine();
//Thread that checks for current price and completes order
validatePrice = new Thread(() =>
{
    Random randomCurrentPriceofStock = new Random();
    while (!stockTrading.StockPurchased)
    {
        stockTrading.currentStockPriceOfXYZ =
randomCurrentPriceofStock.Next(1, 5);
        stockTrading.ValidatePrice();
        Thread.Sleep(1000); // Wait for input before
executing next iteration or else screen will overflow
    }
});
validatePrice.Start();

Console.ReadLine();
```

So, if you noticed before placing new orders, we calling `Reset` method of `ManualResetEvent`, which will ensure that it is reset back to the non-signaled state and any subsequent calls to `Wait()` will be blocked until they receive signal. We will run this application now:

```

Please enter buy price of stock XYZ
Enter price at which you want to buy XYZ (minimum 1, maximum 5
1
3 orders placed, press enter to start stock price matching!!

Current stock price of 4 is not matching with buy price of 1
Current stock price of 1 is matching with buy price of 1
Stock purchased at buy price of 1, Stock order 1 is completed
Stock purchased at buy price of 1, Stock order 3 is completed
Stock purchased at buy price of 1, Stock order 2 is completed

Please enter buy price of stock XYZ
2
2 orders placed, press enter to start stock price matching!!

Current stock price of 3 is not matching with buy price of 2
Current stock price of 3 is not matching with buy price of 2
Current stock price of 1 is not matching with buy price of 2
Current stock price of 1 is not matching with buy price of 2
Current stock price of 3 is not matching with buy price of 2
Current stock price of 4 is not matching with buy price of 2
Current stock price of 3 is not matching with buy price of 2
Current stock price of 2 is matching with buy price of 2
Stock purchased at buy price of 2, Stock order 4 is completed
Stock purchased at buy price of 2, Stock order 5 is completed

```

Figure 7.13: Output of stock trading application using ManualResetEvent

If we do not call `Reset` between orders, they will complete without any price matching. So, a `ManualResetEvent` is helpful in scenarios where all threads need to be unblocked based on an event. There is another signaling construct, `CountdownEvent`, which is the exact opposite of this; that is, multiple threads will send a signal for one signal to process; we will see this in the next section.

CountdownEvent

`CountdownEvent` is a signaling construct that waits for n number of threads before sending a signal to blocked threads. `CountdownEvent` does this by maintaining a count that is used to signal to a waiting thread once it reaches zero.

Taking an analogy of a bus service that can only leave a station only after all the passengers on board the bus (assuming bus service wants to maximize profit hence not sticking to time but gives more importance for all passengers to onboard), the driver will wait till the last passenger onboarded. Here passengers are threads; the bus is the shared resource, and the driver is the waiting thread that waits for all threads to complete.

`CountdownEvent` is initialized by its constructor, which takes an integer as its parameter; this integer input to the constructor is what decides the number of threads that need to be completed before sending a signal to the waiting thread. Initialization will look like below:

```
CountdownEventfileManager = new CountdownEvent(10);
```

In this case, the waiting thread will wait till ten other threads are processed. To achieve this, `CountdownEvent` has the following essential methods:

- `Wait`: Call to this method will make the thread to wait until a signal is received by `CountdownEvent`, that is, typically n number of threads calling `Signal` method
- `Signal`: This method decrements the `currentcount` of `CountdownEvent` when `currentcount` is zero waiting thread will receive signal
- `Reset`: Reset's `currentcount` of `CoutdownEvent` to the value `CountdownEvent` is initialized with.

Let us see this with a simple file download simulator example where we will simulate file downloading and downloading is broken into multiple parallel requests, and finally, when all the parallel requests are completed, parent thread is notified to merge chunks into a single file. Let us create a class called `FileDownloadSimulator` which will have a `CountdownEvent` a `ConcurrentDictionary`:

```
public class FileDownloadSimulator
{
    CountdownEvent fileManager;
    ConcurrentDictionary<int, string> tempFileResponse = new
    ConcurrentDictionary<int, string>();
    public FileDownloadSimulator(int
numberOFThreadsProcessingFileDownload)
    {
        fileManager = new
        CountdownEvent(numberOFThreadsProcessingFileDownload);
    }
}
```

Let us add two methods to this class:

- `SimulateFileDownload`: This method will simulate file download and call `Signal()` method of `CountdownEvent`. Since this simulator, we will just add a text to the concurrent dictionary.
- `FileMerge`: This method will merge all the chunks into one single file and print. This method must wait until all the chunks are downloaded.

These methods will look like the following code:

```
public void SimulateFileDownload(int threadID)
{
```

```
        Thread.Sleep(200);
        tempFileResponse.TryAdd(threadID, $"Line {threadID +
1} of file.\t");
        Console.WriteLine($"Finished processing {threadID}");
        fileManager.Signal();
    }

    public string FileMerge()
    {
        fileManager.Wait();
        Console.WriteLine("Finished Processing, printing
contents");
        StringBuilder fileContents = new StringBuilder();
        for (int i = 0; i < tempFileResponse.Count; i++)
        {
            string output;
            tempFileResponse.TryGetValue(i, out output);
            fileContents.Append(output);
        }
        return fileContents.ToString();
    }
}
```

Now create a console application and add this class to our console application, also simulate parallel threads that will call the `SimulateFileDownload` method and then eventually `FileMerge` method. With this our main class will look like below:

```
static void Main(string[] args)
{
    Console.WriteLine("Welcome to file downloader, please enter
number of parallel threads file download needs to occur");
    int numberOFThreadsProcessingFileDownload = Convert.
ToInt32(Console.ReadLine());
    FileDownloadSimulator fileDownloadSimulator = new
FileDownloadSimulator(numberOFThreadsProcessingFileDownload);
    for (int i = 0; i < numberOFThreadsProcessingFileDownload; i++)
    {
        int captured = i;
```

```
        Thread t = new Thread(() => fileDownloadSimulator.  
SimulateFileDownload(captured));  
        t.Start();  
    }  
    Console.WriteLine(fileDownloadSimulator.FileMerge());  
    Console.ReadLine();  
}
```

Once we run, this application will see an output based on the input on several parallel chunks; in this case, we gave input of 20. So, that will cause `CountdownEvent` to Wait till 20 threads are processed (20 times method is called continuously):

```
Welcome to file downloader, please enter number of parallel threads file download needs to occur  
20  
Finished processing 4  
Finished processing 3  
Finished processing 0  
Finished processing 1  
Finished processing 2  
Finished processing 5  
Finished processing 6  
Finished processing 7  
Finished processing 8  
Finished processing 9  
Finished processing 10  
Finished processing 11  
Finished processing 12  
Finished processing 13  
Finished processing 14  
Finished processing 15  
Finished processing 16  
Finished processing 17  
Finished processing 18  
Finished processing 19  
Finished Processing,printing contents  
Line 1 of file. Line 2 of file. Line 3 of file. Line 4 of file. Line 5 of file. Line 6 of file. Line 7 of file. Line 8 of file.  
Line 9 of file. Line 10 of file. Line 11 of file. Line 12 of file. Line 13 of file.  
Line 14 of file. Line 15 of file. Line 16 of file. Line 17 of file. Line 18 of file.  
Line 19 of file. Line 20 of file.
```

Figure 7.14: Output of parallel file download using CountdownEvent

So, this is a simple scenario where we can take advantage of `CountdownEvent` for signaling between threads. It can be further scaled up to a real-time API call to download a file to speed up file download, or another example could be multiple threads making DB calls and parent thread is waiting on all these DB threads to consolidate (or apply business rules) before sending a response back to the client.

Barrier classes

The barrier is a signaling construct that helps in threads to wait for each other, primarily at a specific point. So, it acts like a barrier where each thread/task executes a piece of code and then waits at a specific point for all the other threads/tasks to finish their execution, once all the threads reach specific waiting point threads are

allowed to proceed with subsequent execution, this also helps in scenarios where there are multiple phases, and all threads need to complete each phase before they can start on the next phase. Barrier also supports post phase execution, which will be executed after all threads complete sending in every phase. A pictorial representation would look like below:

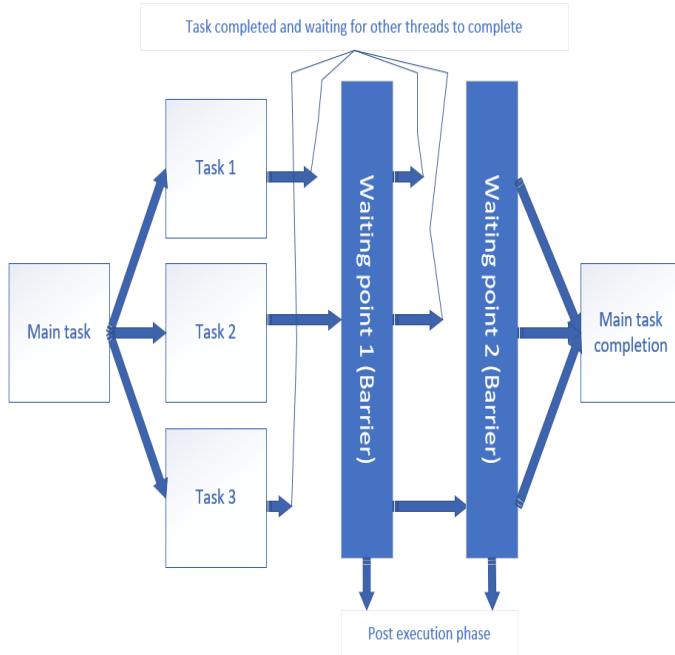


Figure 7.15: Barrier split into three tasks and going through two phases

Let's see this with a simple example where we are building a system to check whether a person is eligible for a home loan, the decision for eligibility is decided by credit score from multiple sources and if the credit score is good from each source then will move to next phase where we look some additional social sources and then approve loan accordingly. Let us start with creating a `HomeLoan` class that has a `Barrier` object and a couple of Boolean fields to track various activities during verification. A parametrized constructor of this class is used to initialize `Barrier` with several participants set to input parameter and a post-execution phase; with this, our class will look like the following code:

```
public class HomeLoan
{
    const int minScore = 150;
    const int maxScore = 1000;
    Barrier barrier;
```

```
public bool HomeLoanStatus { get; set; }

bool creditScoreStatus, socialScoreStatus;

public HomeLoan(int numberOfParticipants)
{
    barrier = new Barrier(numberOfParticipants, (myBarrier) =>
    {
        Console.
WriteLine($"=====");
        Console.WriteLine($"Phase {barrier.CurrentPhaseNumber} finished for all sources");
        Console.
WriteLine($"=====");
    });
    this.HomeLoanStatus = creditScoreStatus = socialScoreStatus
= true;
}
}
```

Add three methods to this class:

- `GetCreditScore` and `GetSocialScore`: Both of them simulate returning a random number, which will be used as a credit score and social score.
- Definition of these methods will look like the following code:

```
int GetCreditScore()
{
    Random rnd = new Random();
    return rnd.Next(minScore, maxScore);
}

int GetSocialScore()
{
    Random rnd = new Random();
    return rnd.Next(minScore, maxScore);
}
```

- `HomeanAnalyzerAsync(string sourceName)`: This method will be split into two phases:
 - **Phase 0:** Credit score evaluation
 - **Phase 1:** Social score evaluation, this phase checks social score only if the credit score is above a specific number or else will exit and set home loan status accordingly

Definition of these methods will look like the following code:

```
public async Task HomeanAnalyzerAsync(string sourceName)
{
    await Task.Factory.StartNew(() =>
    {
        // Start of phase 0
        Console.WriteLine($"Credit score evaluation, phase {barrier.CurrentPhaseNumber}, from source {sourceName} started");
        int creditScore;
        creditScore = GetCreditScore();
        if (creditScore < 200 && creditScoreStatus)
        {
            creditScoreStatus = false;
        }
        // Signal the barrier
        barrier.SignalAndWait();
        // start of phase 1
        Console.WriteLine($"Social score evaluation, phase {barrier.CurrentPhaseNumber}, from source {sourceName} started");
        if (!creditScoreStatus)
        {
            Console.WriteLine($"Bad credit score from source {sourceName}");
            this.HomeLoanStatus = false;
        }
        else
        {
            int socialScore;
```

```
        socialScore = GetSocialScore();
        if (socialScore < 200 && socialScoreStatus)
        {
            Console.WriteLine($"Bad social score from source
{sourceName}");
            socialScoreStatus = false;
            this.HomeLoanStatus = false;
        }
    }
    //signal the barrier
    barrier.SignalAndWait();
});
}
```

Create a console application and add this class. In the `Main` method, take input from the user on the number of sources that are going to be used for home loan evaluation and create tasks for each source to execute them parallel by calling the `HomeanAnalyzerAsync` method. Finally, print the output of home loan approval/rejection. Our `Main` method will look like the following code:

```
static async Task Main(string[] args)
{
    Console.WriteLine("Welcome to home loan analyzer, please
enter number of sources needed for verification");
    int numberofSources = Convert.ToInt32(Console.ReadLine());
    Task[] tasks = new Task[numberofSources];
    HomeLoanhomeLoan = new HomeLoan(numberofSources);
    for (int i=0;i<numberofSources;i++)
    {
        tasks[i] = homeLoan.HomeanAnalyzerAsync(i.ToString());
    }
    await Task.WhenAll(tasks);
    if (homeLoan.HomeLoanStatus)
        Console.WriteLine("Home loan approved");
    else
        Console.WriteLine("Home loan rejected");
```

```

        Console.ReadLine();
    }
}

```

Once we run this application output will look like the following screenshot:

```

Welcome to home loan analyzer, please enter number of sources needed for verification
5
Credit score evaluation, phase 0, from source 1 started
Credit score evaluation, phase 0, from source 0 started
Credit score evaluation, phase 0, from source 3 started
Credit score evaluation, phase 0, from source 2 started
Credit score evaluation, phase 0, from source 4 started
=====
Phase 0 finished for all sources
=====
Social score evaluation, phase 1, from source 0 started
Social score evaluation, phase 1, from source 4 started
Social score evaluation, phase 1, from source 2 started
Social score evaluation, phase 1, from source 3 started
Social score evaluation, phase 1, from source 1 started
=====
Phase 1 finished for all sources
=====
Home loan approved

```

Figure 7.16: Home loan analyzer using Barrier

Some essential facts about **Barrier**:

- All threads waiting on each other including the Main thread
- The barrier is reused, that is, `SignalAndWait` can be called based on the number phases and is irrespective of the value Barrier is initialized with
- Any post phase exception in results in `BarrierPostPhaseException`

In this section, we discussed a signaling construct known as **Barrier** and scenarios where it is useful. Here we have seen how **Barrier** can be used in a situation where we have an extensive processing system and multiple phases and how parallel threads can be used and synchronized. In the next section, we will see any other synchronization context that is available in .NET.

Wait and Pulse

Wait, and **Pulse** are methods available in **Monitor** class that can be used to build a custom signaling construct, that is, if we do not want to use any of the available constructs and build our class that can be used for in thread signaling then **Wait** and **Pulse** methods are way to go:

- **Wait** method blocks the thread
- **Pulse** method sends a signal to the thread that is blocked; **Pulse** can unblock one thread, something like in **AutoResetEvent**

- `PulseAll` sends a signal to all blocked threads, so this can be used to release all threads, something like in `ManualResetEvent`

If we are building a custom signaling construct, then we need to handle all the flags that are needed for thread signaling like current count, reset, initial state, and so on. Also, we need to use some locking constructs to protect shared variables in such class. Although `Wait` and `Pulse` can be used to build our custom signaling construct, we should try to use the ones given by .NET, and if still, they are not solving the problem, then go for `Wait` and `Pulse`.

It concludes signaling constructs where we have seen the usage of `AutoResetEvent`, `ManualResetEvent`, `CountdownEvent`, `Barrier`, and finally, `Wait/Pulse`. Apart from these, there are some more synchronization constructs available in .NET, which we will see briefly in the next section.

Interlocked class

`Interlocked` class is a static class that has methods available to achieve non-blocking synchronization. It is typically used in scenarios where the shared variable has increments or decrements and needs to be locked during concurrency, so instead of wrapping that variable with a lock statement, we can use the `Interlocked` class to modify the values of that variable. The usage of `Interlocked` class is much easier, and it is faster as compared to locks. It has the following primary methods to modify a variable `Increment`—used to increment the value of the variable. For example:

```
int x = 0;  
Interlocked.Increment(ref x); //Value of x changes to 1
```

- `Decrement`: Used to decrement the value of a variable
- `Exchange`: Used to assign a value to a variable

```
int x = 0;
```

```
Interlocked.Exchange(ref x, 10); // Value of x is 10 now
```

- `CompareExchange`: This method takes three parameters, compares first and last parameter and if both are same assigns second parameter to first parameter:

```
int x = 10;
```

```
Interlocked.CompareExchange(ref x, 20, 10); // Value of x is  
20 now
```

The good thing about `interlocked` is it helps in achieving atomicity, that is, it optimizes CPU instructions and execute them as one single instruction (that is, all instructions executed in one single go without context switching) and is mostly used in tandem with other synchronization constructs to achieve simple increment, decrement or assignment.

Volatile class

Volatile is another static class that is used to prefix any variable which enables the variable to be refreshed immediately across processors in the multi-core system. In a typical scenario, the variable is cached across processors, and one processor changing it may not reflect immediately in another processor. The standard lock should solve this problem, but declaring a variable `Volatile` ensures that any write operation performed on the variable is immediately reflected across processors.

The usage of the `Volatile` keyword should be limited and can quickly go wrong as `Volatile` is valid only when a thread performs a single read or write operation on the variable. If a single instruction includes both read and write (like `x++;`) then we should avoid using `Volatile` and `lock` or `Interlocked` class.

Summary

In this chapter:

- We have seen various synchronization constructs
- Starting from exclusive locks to non-exclusive locks
- Looked at various signaling constructs and how signaling can be achieved in .NET using various classes like `Mutex`, `Semaphore`, `AutoResetEvent`, and many more
- Also understood other classes available like `Volatile`, `Interlocked` available in .NET that are used for synchronization.

The samples in this example, developers should be able to choose the right set of synchronization construct that fits their requirement and can implement synchronization in their application accordingly.

In the next chapter, we will see what unit testing is, why do we need unit testing, what is the importance of unit testing in parallel programming, and demonstrates how to write unit tests for parallel and asynchronous programs using XUnit.

Exercise

1. What is the difference between `AutoResetEvent` and `ManualResetEvent`?
2. Replace `CountdownEvent` in the file download example used using other PLINQ and other Parallel classes like `Parallel.For` and `Parallel.ForEach`.
3. Write an application to measure the performance of examples used in `Semaphore` and `SemaphoreSlim` class.
4. What is the difference between `CountdownEvent` and `Barrier`?
5. What is the difference between `Volatile` and `Interlocked` classes?

CHAPTER 8

Unit Testing

Parallel and

Asynchronous

Programs

"With great power comes great responsibility."

In this chapter, we will cover what unit testing, why unit testing is needed, and we will see how to write unit tests for methods that support parallel execution and also for methods that can be executed asynchronously.

Structure

- Unit testing (What, Why and How)
- Unit test for an async program using XUnit
- Unit test for the parallel program using XUnit
- Summary
- Exercise

Objectives

By the end of this chapter reader should be able to understand:

- What is unit testing and why unit tests are needed?
- How to write unit tests for an async method?
- How to write unit tests for parallel method?

Overview

Before we get started on how to do the unit testing, the first thing that we should understand what unit tests are and why unit testing is needed. A unit test is a method that calls a method in our application and validates the response of that method against a predefined value. This predefined value is called **mock data**, and the process of validating mock data with the output of the calling method is called an **assertion**.

For example, if there is a method `Divide` in my application that takes two integers as input and returns division of those methods, a typical unit test for such method will look like the following code:

```
public void TestDivide()
{
    var mathClass = new MathClass();
    int output = 3; //mock data
    var result = mathClass.Divide(6, 2);
    Assert.Equal(output, result);
}
```

In this example, the first we define the mock data, that is, expected output, then after the initialization, we call the actual method and store its result, which is used to compare with the expected output. The idea of writing unit tests is that even though the underlying method (in this case, `Divide` method of `MathClass`) may undergo some changes like a different library can be used to calculate division, and so on, but any of these changes should not change the output of this method.

One way to ensure that a change in method has not broken anything is to execute it manually and validate the output; however, this is error-prone as there could be many scenarios in real-time applications. Another way to do this is to write a unit test(s), which can be executed every time the method is changed to ensure the final output of the method is not changed.

There are many frameworks available to write unit tests as the built-in one comes directly with Visual Studio, then we have third party frameworks like NUnit, XUnit, and many more. More or less, each framework supports writing all kinds of unit tests, so it's upto developers on choosing which Framework they want to use as all the frameworks do support writing any kind of unit tests, including unit tests that support `async/await`. For this book, we will focus on using XUnit; however, all the samples can be written in other frameworks as well.

Basics of unit testing with XUnit

XUnit is a unit testing library that comes with all the necessary methods to unit test our application code. When using XUnit, any method that is annotated with keyword [Fact] becomes a test method. In general, unit tests are created as part of a separate class library project where we add a class file, add a reference to the class that needs to be tested and then create a test method by annotating it with keyword [Fact]. Although unit test classes can be part of the same project as the class that it is testing, it is recommended to make it a separate project for easy maintainability and segregation.

Let us create a class library project and add a simple class file that we are going to test, let us call it **MathClass**. Add a method **Divide** that accepts two parameters and returns division, with this **MathClass** class, will look like the following code:

```
public class MathClass
{
    public int Divide(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new DivideByZeroException();
        }
        else
            return numerator / denominator;
    }
}
```

Now let's add a unit test for this method, to start with let us create a class library project **UnitTests** and add a class, name it **MathClassUnitTest**. Now since we are using XUnit, we need to install the XUnit package. Open package manager console and run the following command, as shown in *Figure 8.1*:

Install-Package xunit



Figure 8.1: Install XUnit through package manager console

Also, we need to install an XUnit runner package that will help to run the test case through a Visual Studio test runner. For this run the following command:

Install-Package xunit.runner.visualstudio

Now follow the steps to add a unit test:

1. Add reference of `MathClass` by referencing `Calculator` project to unit test project
2. Add a method `TestDivide` that creates an object of `MathClass` and call `Divide` method
3. To verify output with the expected value, we will make use of `Assert` class of XUnit
4. Annotate the method with keyword `[Fact]`

With this unit test class will look like below:

```
using Calculator;
using Xunit;
public class MathClassUnitTest
{
    [Fact]
    public void TestDivide()
    {
        var mathClass = new MathClass();
        int output = 3; //mock data
        var result = mathClass.Divide(6, 2);
        Assert.Equal(output, result);
    }
}
```

Here we are following the AAA technique (Arrange, Act and Assert), where we start with initializing data and then invoke the method and eventually validate output with the expected result. In Visual Studio 2019, to execute a test, the method application needs to be successfully built. So, build the application, and once it is successfully built, there are multiple ways, as mentioned below, to run a unit test.

Executing unit tests

Visual Studio Test Explorer lists down all the unit tests available in our solution. Navigate to **Test Explorer** through **View | Test Explorer** where we can see our test method as shown in *Figure 8.2*:

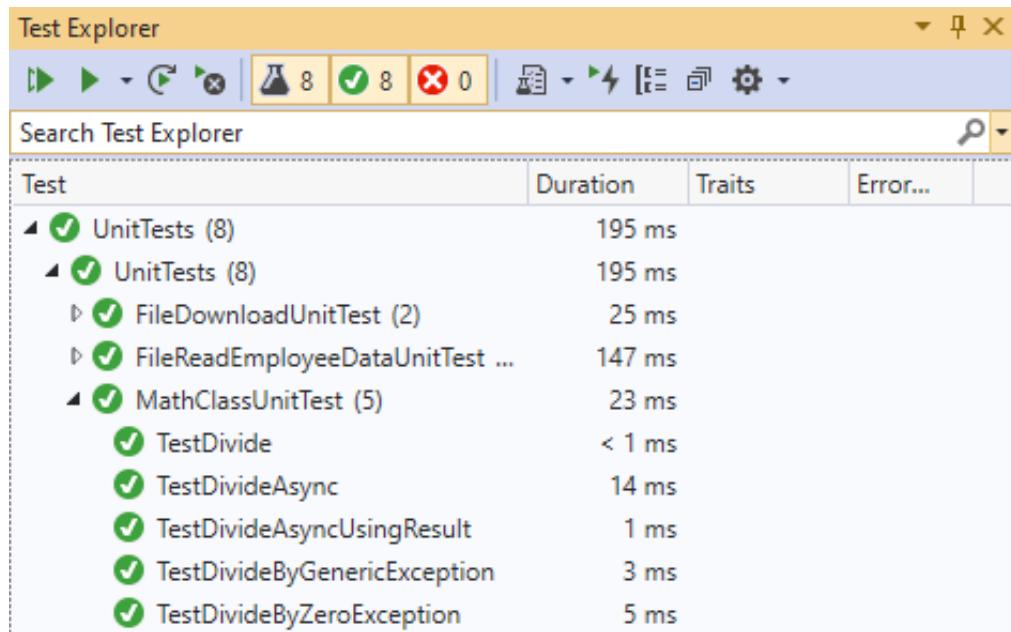


Figure 8.2: Test Explorer

Now right click on the test divide method and click **Run** to execute the test, if the test passes blue icon will turn green if the test fails then it will turn red with an error message at the bottom of the Test Explorer. Similarly, we can debug the test method.

Another way to run/debug a unit test is by clicking on the blue icon just above the unit test method, as shown in *Figure 8.3*:

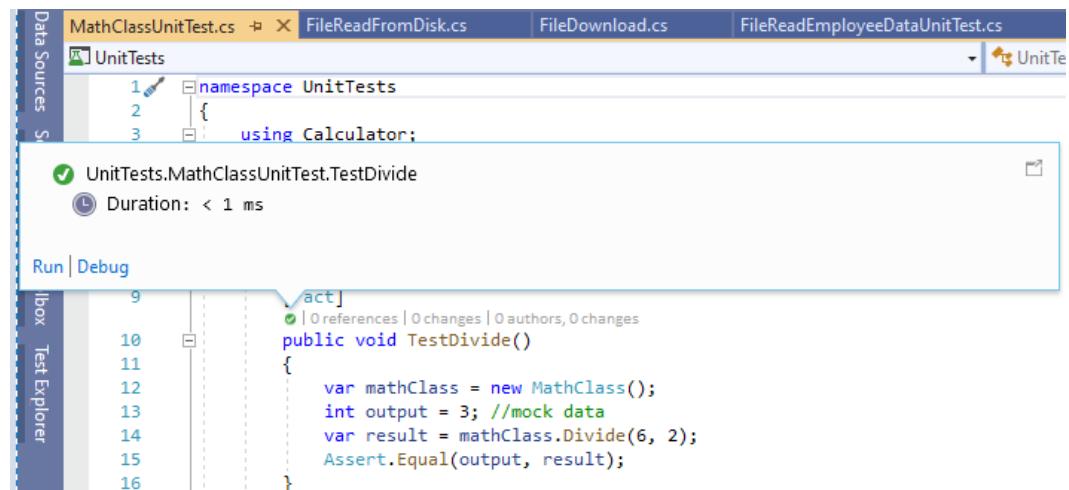


Figure 8.3: Run/Debug unit test

Another option to run unit tests is through the developer command prompt by passing `unit test dll` as a parameter to `vstest.console.exe`. So, in our case, open the VS 2019 developer command prompt and run `vstest.console.exe UnitTests.dll` to run all the unit tests. The output will look something like in *Figure 8.4*:

```
Microsoft (R) Test Execution Command Line Tool Version 16.3.0-preview-20190715-02
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
[xUnit.net 00:00:00.00] xUnit.net VSTest Adapter v2.4.1 (64-bit .NET Core 3.0.0)
[xUnit.net 00:00:01.93] Discovering: CalculatorUnitTest
[xUnit.net 00:00:01.98] Discovered: CalculatorUnitTest
[xUnit.net 00:00:01.98] Starting: CalculatorUnitTest
[xUnit.net 00:00:02.10] Finished: CalculatorUnitTest
  ✓ CalculatorUnitTest.MathClassUnitTest.TestDivide [5ms]

Test Run Successful.
Total tests: 1
  Passed: 1
Total time: 3.7392 Seconds
```

Figure 8.4: Run unit test through the command line

`vstest.console.exe` also gives many options to execute specific tests, and so on and is very useful if there is a need to execute unit tests without the Visual Studio. A most common scenario is while building continuous integration/continuous deployment (CI/CD) pipelines where one of the tasks is to execute all unit tests.

In this section, we covered how to create and configure unit test projects and various options to run unit tests. In the next sections, we will focus on how to write unit tests for asynchronous methods and parallel methods.

IntelliTest

IntelliTest is one of the testing tools available in Visual Studio 2015 Enterprise editions onwards, which helps in auto-generation of unit tests. It is available by right-clicking on any public method within a class where it gives an option to generate one or more-unit tests, as shown in *Figure 8.5*:

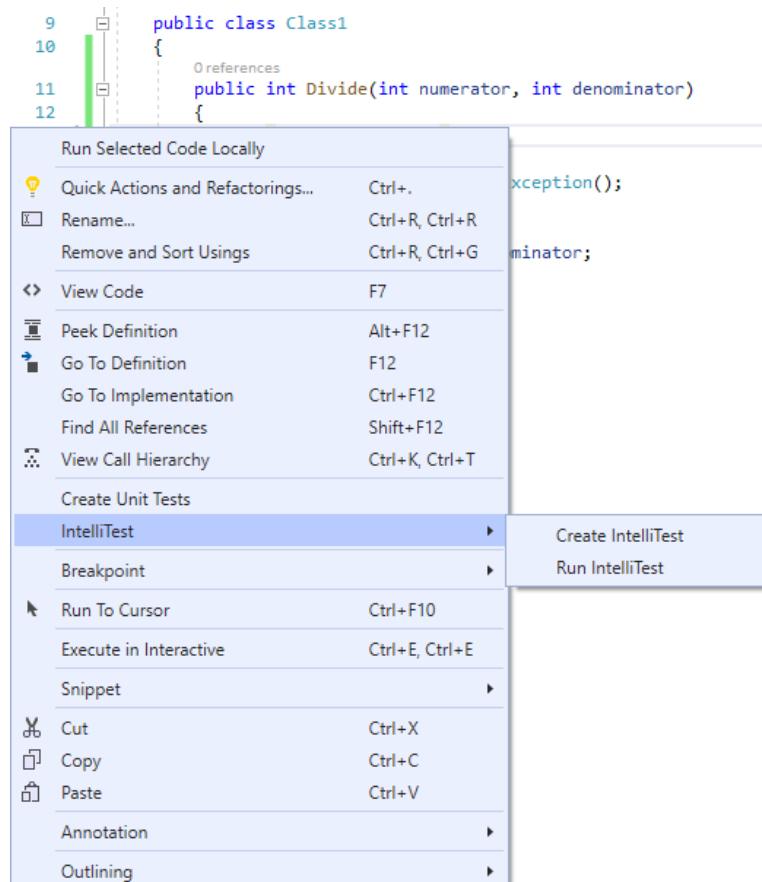


Figure 8.5: IntelliTest in the context menu

We can click on **Run IntelliTest**, which shows output in **Exploration Results** of the various combination that are generated and how many of them passed. These can be further saved directly by clicking on **Save** in **Exploration Results** window, as shown in *Figure 8.6*:

IntelliTest Exploration Results - stopped						
Class1.Divide(Int32, Int32)		target	numerator	denominator	result(target)	result
1	new Class1()	0	0			DivideByZeroException
2	new Class1()	0	1	new Class1()	0	OverflowException
3	new Class1()	int.MinValue	-1			Attempted to divide by zero.

Figure 8.6: IntelliTest Exploration Results window

IntelliTest is an excellent feature of Visual Studio, which helps in generating and maintaining ever-changing code; however, at this point, it doesn't have support for .NET Core and is scheduled for future release.

Live Unit Testing

Live Unit Testing is another feature available in Visual Studio Enterprise editions, which, when enabled, continuously runs our tests in background and reports for any failures. It can be enabled through **Tools | Options | Live Unit Testing|General**:

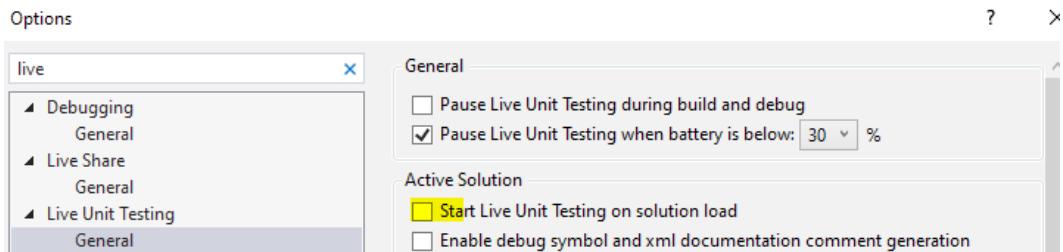


Figure 8.7: Live unit testing

Live Unit Testing also continuously updates code coverage metrics along with running unit tests in the background and is a very productive feature that monitors failures in unit tests and reports them accordingly. Live Unit Testing is supported in both .NET Framework and .NET Core projects.

Unit test async methods

Unit Testing Asynchronous methods isn't as straight forward as Unit Testing Synchronous methods because asynchronous methods aren't completed in one single call and if we test asynchronous methods like synchronous methods our test method won't wait for the completion of asynchronous method and may end up asserting even before asynchronous method completion.

Fortunately, with XUnit writing unit tests is as easy writing any asynchronous method; that is, an asynchronous method that needs to be unit tested can be prefixed using keyword `async`. Let us see this with a simple example by creating a method that takes two parameters and returns division of those parameters (same as the previous one) asynchronously. So, let's add a new method `DivideAsync` to `MathClassUnitTest` class and let's wrap `Divide` method into a `Task` and `await` on that as shown below:

```
public async Task<int> DivideAsync(int numerator, int denominator)
{
    var t = Task.Run(() =>
    {
        return Divide(numerator, denominator);
    });
}
```

```

});  

return await t;  

}

```

Now add a test method `TestDivideAsync` in `MathClassUnitTest`, since we are writing unit test for sync method to assert the output we need to ensure that method execution is completed before assert and the way to do that is nothing different than calling an asynchronous method, that is, to prefix the call to an asynchronous method by `await`. Since we are waiting on one of the method unit test method's returns type would be `async Task` instead of void. So, our method definition will look like the following:

```

[Fact]  

public async Task TestDivideAsync()  

{  

    var mathClass = new MathClass();  

    var result = await mathClass.DivideAsync(6, 2);  

    Assert.Equal(3, result);  

}

```

Build the application and run the unit test, and it should pass, as shown in *Figure 8.8*:

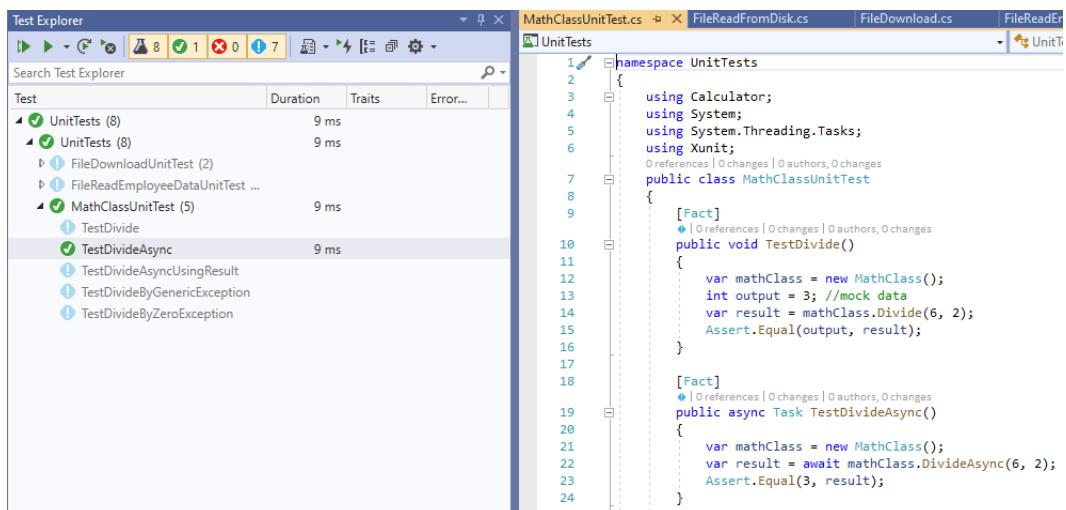


Figure 8.8: Unit test for the async method

Let's tweak the `DivideAsync` method; that is, the method will still divide; however, we will use the `Math` library from framework, and this should not cause any change in the unit test method. With this, our method will look like below:

```
public int Divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        throw new DivideByZeroException();
    }
    else
    {
        int remainder;
        return Math.DivRem(numerator, denominator, out
remainder);
    }
}
```

After this change running `TestDivideAsync` still passes as the only implementation of the method is changed, but the intent remains the same. It is the most significant advantage of writing a unit test where we can ensure that any change to method implementation hasn't broken any of the existing behavior.

Since XUnit support await we can write unit tests that can await with ease; however, if we end up using a testing framework that doesn't support await in unit tests, then we need to follow the technique of calling the asynchronous method from the synchronous method, that is, by using `GetAwaiter().GetResult()`. So, let us add another test that synchronously calls `DivideAsync` method as shown in the following code:

```
[Fact]
public void TestDivideAsyncUsingResult()
{
    var mathClass = new MathClass();
    var result = mathClass.DivideAsync(6, 2).GetAwaiter().
GetResult();
    Assert.Equal(3, result);
}
```

Once we run this test, it will pass; however, this way is not at all recommended as this will possibly cause a deadlock if we are mocking library code. So to be on the safer side, we should use `ConfigureAwait(false)`, that is, the return statement in our `DivideAsync` method should be changed to the following code line:

```
return await t.ConfigureAwait(false);
```

It will ensure that any consumer of our library does not complain of deadlock while consuming it.

Unit test exceptions in async methods

It is a good practice that whenever we write unit tests, they should be written for both positive and negative scenarios, especially exception cases. When a handled exception occurs in the application, our unit test should have the capability to assert against exception.

XUnit gives various overloads of the `Throw` method along with the asynchronous version to assert against any exception. Going back to our divide example since we already handled divide by zero exception, let us write a unit test for this scenario. In this unit test, we will pass denominator as 0 and expected output would be a divide by zero exception, so our unit test will look like the following code block:

```
[Fact]
public async Task TestDivideByZeroException()
{
    var mathClass = new MathClass();
    var result = mathClass.DivideAsync(6, 0);
    await Assert.ThrowsAsync<DivideByZeroException>(async () =>
await result);
}
```

Notice that we are waiting on `ThrowAsync` as if we do not await this test will always pass irrespective of the exception, the reason being the same as not awaiting an asynchronous method.

In this case, we handled a particular exception; however, if we want to handle the generic exception, then XUnit provides `ThrowAnyAsync`, which can receive an exception and pass test case accordingly. So let us tweak `Divide` method a little to throw another exception based on some condition say when the denominator is 1. Adding this conditioning method will look like below:

```
public int Divide(int numerator, int? denominator)
{
    if (denominator == 0)
    {
        throw new DivideByZeroException();
    }
    else if (!denominator.HasValue)
```

```
    {
        throw new ArgumentNullException ();
    }
    else
    {
        int remainder;
        return Math.DivRem(numerator, denominator.Value, out
remainder);
    }
}
```

Now add a unit test that will receive `ArgumentNullException` exception using `ThrowAnyAsync` which looks like the following code:

```
[Fact]
public async Task TestDivideByGenericException()
{
    var mathClass = new MathClass();
    var result = mathClass.DivideAsync(6, null);
    await Assert.ThrowsAnyAsync<Exception>(async () => await
result);
}
```

Build and run this test case, and it will pass. This method will pass in either of the exceptions, that is, `DivideByZeroException` or `ArgumentNullException`. Just like `ThrowAsync`, `ThrowAnyAsync` also needs to be awaited, or else you end up with a test that is always passing.

Unit test async method using mock data

In most of the enterprise applications, unit tests are primarily around service layer classes where will have outbound calls like a database call or loading file in memory and many more. However, unit tests aren't supposed to make outbound calls; instead, they should create mock data for all the outbound calls involved and then validate the business logic method that is unit tested for.

Let us see this with a simple example where we have a method that downloads a file from web asynchronously and then reads file content, applies some logic (in our case, we will do a string reversal), and send back response. In this case, file downloading is an external call, and for unit testing, it should mock with some predefined response.

Add a new class library project; let us call it `FileIO`. Let us start with adding a class `FileDownload`, and we will use `HttpClient` of .NET which will be initialized through the constructor, this class will look like the following code:

```
namespace FileIO
{
    public class FileDownload
    {
        HttpClient _client;

        public FileDownload(HttpClient client)
        {
            if (client != null)
                _client = client;
            else
                _client = new HttpClient();
        }
    }
}
```

Add a method `DownloadFileAsync`; we will use the `GetAsync` method of it to download the file. Once it is downloaded, use `ReadAsStringAsync` to retrieve data and apply string reversal before returning. This method will look like below:

```
public async Task<string> DownloadFileAsync()
{
    string url = "https://github.com/Ravindra-a/largefile/blob/
master/README.md"; //Replace this with any URL

    using (HttpResponseMessage response = await _client.
GetAsync(url)) // Should mock GetAsync for unit tests
    {
        if (response.IsSuccessStatusCode)
        {
            string result = await response.Content.
ReadAsStringAsync();

            // Now reverse this string - In enterprise
            application this will be some business logic
        }
    }
}
```

```
        StringBuilder reverseString = new StringBuilder();
        for (int i = result.Length - 1; i >= 0; i--)
        {
            reverseString.Append(result[i]);
        }
        return reverseString.ToString();
    }
    else if (response.StatusCode == HttpStatusCode.NotFound)
    {
        throw new FileNotFoundException();
    }
    throw new Exception(); //For all other stauts codes
}
}
```

Now let us write a unit test for this method where the focus is on mocking response from `GetAsync` and validating string reversal logic. So before even starting unit tests, we need to create mock objects for all the dependencies, and looking at our `DownloadFileAsync` method, we have only one outbound call, which is `GetAsync` of `HttpClient` class, so we somehow need to create a mock object of `HttpClient`.

Mock objects help in mimicking the behavior just like a real-time call, however instead of returning actual data mock objects will return the response as per dummy data we define, to do this we can use many frameworks like Moq, Fake it easy, and many more, however in this example we will create `HttpClient` object (also as it is only one single mock object we do not need a full-fledged framework)using its constructor that takes `HttpMessageHandler`, for this `HttpMessageHandler` we will create a mock object. Since the `SendAsync` method of `HttpMessageHandler` allows us to customize the response, we can use it in a way that will help us to customize the behavior of `HttpClient`.

So let's add a new class `FakeHttpMsgHandler` to unit test project which is inherited from `HttpMessageHandler` and implement `SendAsync` as shown in the following code:

```
public class FakeHttpMsgHandler : HttpMessageHandler
{
    private HttpResponseMessage _response;

    //Constructor
```

```
public FakeHttpMsgHandler(HttpResponseMessage response)
{
    _response = response;
}

protected override Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, System.Threading.CancellationToken
cancellationToken)
{
    var taskCompletionSource = new
TaskCompletionSource<HttpResponseMessage>();

taskCompletionSource.SetResult(_response);

return taskCompletionSource.Task;
}
}
```

Now add a unit test class `FileDownloadUnitTest` and add a reference to the `FileIO` project. Add a unit test method `DownloadFileSuccess`, as shown below:

```
[Fact]
public async Task DownloadFileSuccess()
{
    // Dummy response
    HttpResponseMessage mockResponse = new HttpResponseMessage
    {
        StatusCode = HttpStatusCode.OK,
        Content = new StringContent("Response from fake httpclient")
    };
    var msgeHandler = new FakeHttpMsgHandler(mockResponse);
    var httpClient = new HttpClient(msgeHandler);
    var fileDownloadObj = new FileIO.FileDownload(httpClient);

    string expectedResult = "Response from fake httpclient";
    //string reversal logic
```

```

        char[] charArray = expectedResult.ToCharArray();
        Array.Reverse(charArray);
        //Call to method
        var result = await fileDownloadObj.DownloadFileAsync();
        Assert.Equal(charArray.Length, result.Length); //assertion
        Assert.Equal(new string(charArray), result); //assertion
    }
}

```

Here we are calling our file download method, which will respond with the mock response object, so the content will be "Response from fake httpclient" which will be reversed as per business rules within the DownloadFileAsync method. Finally, we are asserting that in our unit test. Once we run this test, it will pass as shown in the following screenshot:

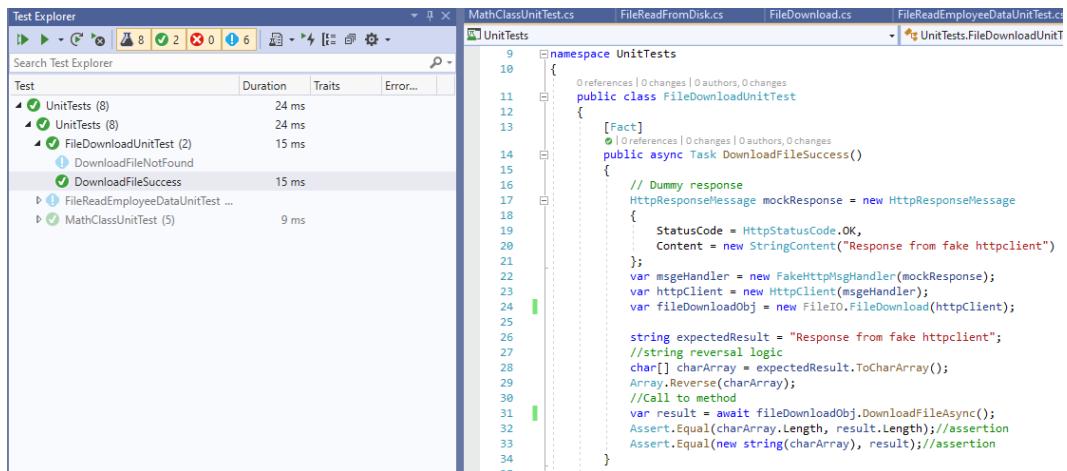


Figure 8.9: Unit test for the async method

Now let us add a negative test case where the response from GetAsync is **404 (not found)**, so the first thing we need to change is the mock response, which is sending HTTPstatus code 404. Then we will use ThrowAsync to assert response against specific exceptions, in this case, `FileNotFoundException`. With this, our unit test will look like below:

```

[Fact]
public async Task DownloadFileNotFound()
{
    // Dummy response
    HttpResponseMessage mockResponse = new HttpResponseMessage

```

```
{  
    StatusCode = HttpStatusCode.NotFound,  
};  
var msgeHandler = new FakeHttpMsgHandler(mockResponse);  
var httpClient = new HttpClient(msgeHandler);  
var fileDownloadObj = new FileIO.FileDownload(httpClient);  
  
var result = fileDownloadObj.DownloadFileAsync();  
await Assert.ThrowsAsync<FileNotFoundException>(async () =>  
await result);  
}
```

So here we ensured that the mock `HttpResponseMessage` object is returning a 404 and verified if our method is returning appropriate exception, which is asserted through using XUnit's `ThrowAsync`. This way, we can add more unit tests. So, in this section, we covered how to create mock objects in case of dependencies so that our unit test focuses on validating business logic.

Unit test for parallel methods

Till now, we have seen various scenarios around unit testing asynchronous methods, in this section, we will see how to unit test when there is parallel code or data parallelism, that is, say if there is a method that uses parallel loops.

In TPL, as we know, we have `Parallel.For` and `Parallel.ForEach` which helps us in executing a method concurrently without bothering about creating threads and manage them, however with parallel code the thing that needs to be managed is handling data-parallel, that is, shared resources should be synchronized either through synchronization constructs like locks, semaphores, and many more or through the data structures available for parallelism like `ConcurrentBag`, `BlockingCollection`, and so on.

In either case from a unit testing standpoint, parallel execution of code does not change much on how unit tests are written, but they are necessary as it helps to test the concurrency logic and validate that shared resources aren't corrupted. Let us see this with an example where we load a huge file that holds employee data

(employee ID and bonus); each employee record is separate by a new line. This file will typically look something like the following screenshot:

```
1 Employee id -2088553071 Bonus -14392
2 Employee id -220105603 Bonus -130140
3 Employee id -1248110392 Bonus -177949
4 Employee id -1587500754 Bonus -86477
```

Figure 8.10: Employee file

We will load this into memory and then parallel process each record, apply some business rules and return the final output, which will be a subset of the list of employees based on the filtering applied in business rules. So, to our `FileIO` class library project let us add an employee class and add the following contents:

```
public class Employee
{
    public int EmployeeID { get; set; }
    public int Bonus { get; set; }
}
```

Now to read a file, we will use a `StreamReader` class, however as we would be writing unit tests, so we need to mock this `StreamReader` such that mock data that is configured in unit tests is returned instead of reading a file from disk. So, we will add an interface `I.FileReader` which will be used to read the file using `StreamReader` and also used later to mock, let's us add an interface `I.FileReader` that looks like the following code:

```
public interface I.FileReader
{
    StreamReader Get.FileReader(string filePath);
}
```

The advantage of using the interface is that it can be mocked very well with any mocking framework. We can also go ahead and do dependency injection, but for now, to keep it simple and focus on unit testing, we will not do that.

Now add a class that implements this interface which gives us an object of `StreamReader`, let us call this class `FileStreamReader`, and it will look like below:

```
public class FileStreamReader : I.FileReader
{
    public StreamReader Get.FileReader(string filePath)
```

```

    {
        return new StreamReader(filePath);
    }
}

```

Now let us add our class where we read a file with the list of employees, process records parallel and apply business rules, let us call this class `FileReadFromDisk` and add the following code to it:

```

public class FileReadFromDisk
{
    private readonly IFileReader _streamReader;

    //Thread safe collection to store exceptions occurred during
    parallel processing
    ConcurrentBag<Exception> errors = new
    ConcurrentBag<Exception>();

    public FileReadFromDisk(IFileReader streamReader)
    {
        this._streamReader = streamReader;
    }
}

```

Then we will create two methods that return **Task**:

1. To read data from file and load into a list of employees. Here we will use something called `BlockingCollection`, which is a thread-safe collection and allows us to read and write to collection concurrently without corrupting collection. We will read line by line and add it to this collection, later we will read from this collection concurrently. This method will look like below:

```

public Task ReadDataFromFile(string filePath,
    BlockingCollection<string> employeeData)
{
    return Task.Factory.StartNew(() =>
    {
        using (StreamReader sr = this._streamReader.
Get.FileReader(filePath))
        {

```

```
        while (!sr.EndOfStream)
    {
        employeeData.Add(sr.ReadLine());
    }
}

// Notify consumers that addition is completed
employeeData.CompleteAdding();
});

}
```

2. To serialize data that is loaded in `employeeData` collection into a collection of `Employee` type, here we will use `Parallel.ForEach` to iterate through `employeeData` collection and load into the collection of `Employee` type. For the collection of `Employee` type, we will use `ConcurrentBag` as it is a thread-safe collection and allows us to add data concurrently without corrupting the list. This method will look like the following code:

```
private Task
SerializeEmployeeData(BlockingCollection<string> employeeData,
ConcurrentBag<Employee> employeeDetails)
{
    return Task.Factory.StartNew(() =>
{
    try
{
    Parallel.ForEach(employeeData.
GetConsumingEnumerable(), line =>
{
    // String manipulation
    string[] lineFields = line.Split('\t');
    int employeeID, bonus;
    int.TryParse(lineFields[1].
Substring(lineFields[1].IndexOf('-') + 1), out employeeID);
    int.TryParse(lineFields[2].
Substring(lineFields[2].IndexOf('-') + 1), out bonus);
    employeeDetails.Add(new Employee {
EmployeeID = employeeID, Bonus = bonus });
}
}
}
}
}
```

```
});  
  
}  
catch (Exception ex)  
{  
    errors.Add(ex);  
}  
});  
});
```

In the above method, we are also doing exception handling as usual in `Parallel.ForEach` loop, if there is an exception in one of the iterations, it won't process any of the subsequent iterations; however, what we want to do here is to handle the exception for every iteration so that we can process all iterations. At the end of this method, we will handle any raised exceptions accordingly; here, we are using `ConcurrentBag` collection of type `Exceptions` that was declared earlier to accumulate exceptions during each iteration. Again the reason to use `ConcurrentBag` is to achieve thread-safety.

Now add a method `ReadFileandProcessTask` that will take two parameters:

1. **filePath as a string:** This is the path of the file that needs to be loaded
2. **bonusAmountRule as int:** This is a variable that is used in business rules to filter employees list

This method will further call the tasks defined above and will implement business rules to filter out the employee list. Code for this method will look like below:

```
public List<Employee> ReadFileandProcessTask(string filePath, int  
bonusAmountRule)  
{  
  
    // Using ConcurrentBag for thread safety  
    var employeeDetails = new ConcurrentBag<Employee>();  
  
    // Blocking collection so that multiple consumers do not end  
    up corrupting data  
    var employeeData = new BlockingCollection<string>();  
    // Single Producer  
    var readLines = ReadDataFromFile(filePath, employeeData);
```

```
// Multiple Consumers
var processLines = SerializeEmployeeData(employeeData,
employeeDetails);

Task.WaitAll(readLines, processLines);

// Throw the exceptions here after the loop completes.
if (errors.Count > 0)
{
    throw new AggregateException(errors);
}

//Business logic - Get all users with bonus greater than
50000
return employeeDetails.Where(x => x.Bonus >=
bonusAmountRule).ToList();
}
```

If we look closely, we are using producer and consumers pattern here where one producer is reading the file, and multiple consumers are notified once the producer completes its tasks.

If we do a build at this stage, the `FileIO` class library should build successfully.

Now let's add unit tests for `ReadFileandProcessTask`, but before we start writing unit tests, let us add a mocking Framework that we planned to use, which is MOQ. So open **Package Manager Console**, under **Default Project**: Select the **UnitTests** project and run the following command (see *Figure 8.11*):

Install-Package Moq

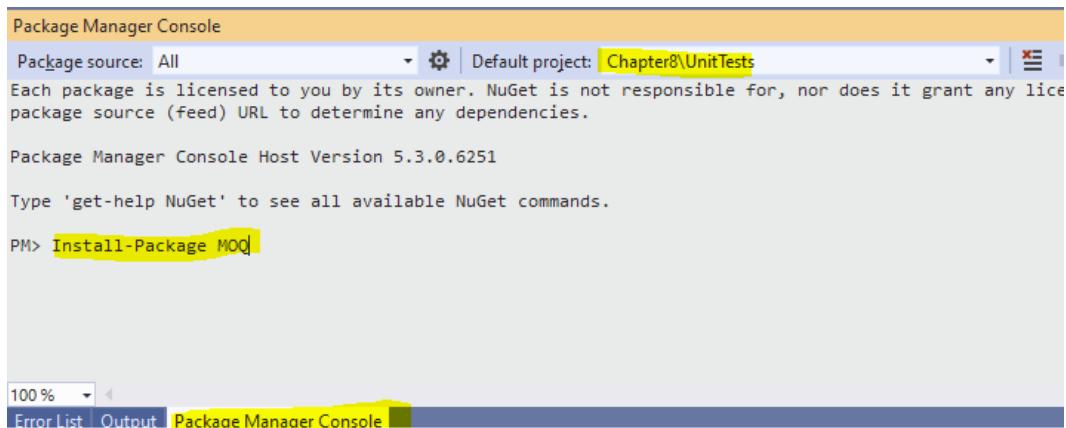


Figure 8.11: Package Manager Console command to install the Moq library

Add a class `FileReadEmployeeDataUnitTest` to `UnitTests` project and add a first unit test, which will be around testing the logic in the application, that is, pass a list of mock employee data and see if we are getting appropriate filter results. So first add a private method that will return mock data, code of that method will look like below:

```
private StringBuilder GetMockFileData ()
{
    // We will follow the pattern used for manipulation in
    // SerializeEmployeeData i.e. data separate by tab
    StringBuilder mockFileData = new StringBuilder();
    mockFileData.AppendLine("1 Employee id -1      Bonus
-14392");
    mockFileData.AppendLine("2 Employee id -2      Bonus
-130140");
    mockFileData.AppendLine("3 Employee id -3      Bonus
-177949");
    mockFileData.AppendLine("4 Employee id -4      Bonus
-86477");
    mockFileData.AppendLine("5 Employee id -5      Bonus
-202725");
    mockFileData.AppendLine("6 Employee id -6      Bonus
-203595");
    mockFileData.AppendLine("7 Employee id -7      Bonus
-43698");
    return mockFileData;
}
```

Now add the unit test method in which we will first create a mock object of file stream and use `MemoryStream` to load mock content into a stream and return an object of `StreamReader` that takes this `MemoryStream` as a parameter. The Moq library gives a method called `Setup` which is used to return mock data when any particular method is called, that is, in `ReadFileandProcessTask` we want to return mock data from `GetMockFileData` when `StreamReader` reads the file and calls methods like `ReadLine` and `EndOfFile`, so we will use `Setup` method of Moq library to return a `StreamReader` that loads mock data we created.

Once this is done we will call the `ReadFileandProcessTask` followed by asserts, with this code for the unit test will look like below:

```
[Fact]
public void EmployeeDetailsEmployeesWithHigherBonusFound()
{
    //Setup mocking data
    string mockPath = "mockPath";
    StringBuilder content = GetMockFileData();
    MemoryStream ms = new MemoryStream(Encoding.UTF8.
GetBytes(content.ToString()));
    Mock<I.FileReader> reader = new Mock<I.FileReader>();
    //Using Moq to respond with mock data for file stream
    reader.Setup(sr => sr.FileReader(mockPath)).Returns(new
StreamReader(ms));
    FileReadFromDisk sut = new FileReadFromDisk(reader.Object);
    //Call the app
    List<Employee> employeesWithHigherBonus = sut.
    ReadFileandProcessTask(mockPath, 200000);
    //Assert
    Assert.NotNull(employeesWithHigherBonus);
    Assert.Equal(2, employeesWithHigherBonus.Count);
}
```

Once we run this, we will see that the test will pass, as shown in *Figure 8.12* as our mock data has two records that have a bonus higher than 200000:

```
[Fact]
● 0 references | 0 changes | 0 authors, 0 changes
public void EmployeeDetailsEmployeesWithHigherBonusFound()
{
    //Setup mocking data
    string mockPath = "mockPath";
    StringBuilder content = GetMockFileData();
    MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(content.ToString()));
    Mock<I.FileReader> reader = new Mock<I.FileReader>();
    //Using Moq to respond with mock data for file stream
    reader.Setup(sr => sr.GetFileReader(mockPath)).Returns(new StreamReader(ms));
    FileReadFromDisk sut = new FileReadFromDisk(reader.Object);
    //Call the app
    List<Employee> employeesWithHigherBonus = sut.ReadFileandProcessTask(mockPath, 200000);
    //Assert
    Assert.NotNull(employeesWithHigherBonus);
    Assert.Equal(2, employeesWithHigherBonus.Count);
}

[Fact]
● 0 references | 0 changes | 0 authors, 0 changes
public void EmployeeDetailsEmployeesWithHigherBonusNot()
{
}
```

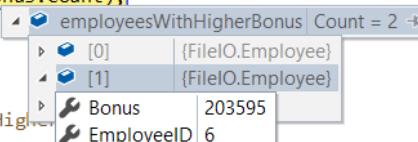


Figure 8.12: Unit test in debug mode with output

Now let us add another unit test where our method returns an empty object, so for this, we will pass `String.Empty` to our `MemoryStream` and rest remains the same as the previous test, and this time, we will use `Assert.Empty` of XUnit as there won't be any employees in the response from `ReadFileandProcessTask`. With this, our unit test would look like below:

```
[Fact]
public void EmployeeDetailsEmployeesWithHigherBonusNotFound()
{
    //Setup mocking data
    string mockPath = "mockPath";
    MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(String.Empty));
    Mock<I.FileReader> reader = new Mock<I.FileReader>();
    //Using Moq to respond with mock data for file stream
    reader.Setup(sr => sr.GetFileReader(mockPath)).Returns(new StreamReader(ms));
    FileReadFromDisk sut = new FileReadFromDisk(reader.Object);
    //Call the app
    List<Employee> employeesWithHigherBonus = sut.ReadFileandProcessTask(mockPath, 0);
```

```

    //Assert
    Assert.Empty(employeesWithHigherBonus);
}

```

Once we run this unit test, it will pass, as shown in *Figure 8.13*:

```

[Fact]
public void EmployeeDetailsEmployeesWithHigherBonusNotFound()
{
    //Setup mocking data
    string mockPath = "mockPath";
    MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(String.Empty));
    Mock<IFileReader> reader = new Mock<IFileReader>();
    //Using Moq to respond with mock data for file stream
    reader.Setup(sr => sr.GetFileReader(mockPath)).Returns(new StreamReader(ms));
    FileReadFromDisk sut = new FileReadFromDisk(reader.Object);
    //Call the app
    List<Employee> employeesWithHigherBonus = sut.ReadFileandProcessTask(mockPath, 0);
    //Assert
    Assert.Empty(employeesWithHigherBonus);
}

```

Figure 8.13: Unit test in debug mode with output

Now let us try a scenario where the parallel loop throws an exception and write a unit test for that. To simulate this scenario, we will create mock data that breaks the pattern of our records, which will throw an exception, and since we are continuing the parallel loop and accumulating exceptions in `AggregateException`, we need to **Assert** output with `AggregateException` using `Throw` method of XUnit. Code for this unit test will look like below:

```

[Fact]
public void EmployeeDetailsProcessingFailed()
{
    //Setup mocking data
    StringBuilder mockFileExceptionData = new StringBuilder();
    // Record to throw exception
    mockFileExceptionData.AppendLine("Exception record");
    mockFileExceptionData.AppendLine("1      Employee id -1
Bonus -14392");
    string mockPath = "mockPath";
    MemoryStream ms = new MemoryStream(Encoding.UTF8.
GetBytes(mockFileExceptionData.ToString()));

```

```
Mock<I.FileReader> reader = new Mock<I.FileReader>();
reader.Setup(sr => sr.GetFileReader(mockPath)).Returns(new
StreamReader(ms));
//Call the app
FileReadFromDisk sut = new FileReadFromDisk(reader.Object);
//Assert
var ex = Assert.Throws<AggregateException>(() => sut.
ReadFileandProcessTask(mockPath, 10000));
Assert.Single(ex.InnerExceptions);
//Asserting inner exception
Assert.Equal((new IndexOutOfRangeException()).GetType().Name,
ex.Flatten().InnerExceptions[0].GetType().Name);
}
```

So in this, we are asserting against the `Throw` method of `Assert` and using `Flatten` method of `AggregateException` to getting the actual exception and asserting. As others this once you run this test, it should pass.

So in this section, we primarily covered the things that need to be taken care of while unit testing parallels methods and went through small real-time scenarios and wrote unit tests for it.

Note: If you want to test this class in real-time, they can use the following code to generate a dummy file:

```
if (!File.Exists(filePath))
{
    Random rand = new Random();
    using (FileStream fs = new FileStream(filePath, FileMode.
Create))
    {
        using (StreamWriter sw = new StreamWriter(fs,
System.Text.Encoding.Default))
        {
            for (int i = 1; i <= 1000000; i++)
            {
                sw.WriteLine($"{i}\tEmployee id -{rand.Next()}\t
Bonus -{rand.Next() / 10000} {Environment.NewLine}");
            }
        }
    }
}
```

```
    }  
}  
}
```

It is code to generate a large file with dummy employee details separated by a tab.

Unit test async void methods

There is no better way to test the `async void` method because it's incorrect to write a method with the signature `async void`. As discussed in an earlier chapter, any method with a signature `async void` will run into below two problems:

1. Do not handle exceptions correctly and can lead to crashing the process
2. Cannot be unit tested

So, the recommendations would be to change all methods with a signature `async void` to `async Task` as that solves both the problems mentioned above. For event handlers, there are ways like using callbacks to unit test the code.

Summary

Writing unit tests is one of the critical aspects of application development as it helps in maintainability, encourages loosely coupled design, helps in extending existing features without breaking existing functionality, and many more advantages.

As we have seen in this chapter, writing unit tests for asynchronous/parallel methods is not much different than writing unit tests for synchronous methods it's recommended to write unit tests and make it a mandatory step in your application development.

In this chapter:

- We covered the benefits unit tests, various frameworks available for unit tests.
- How to write unit tests for asynchronous methods, write unit tests for both positive and exception handling scenarios using XUnit?
- How to write unit tests for parallel methods, write unit tests for both positive and exception handling scenarios using XUnit?
- How to mock data using Moq?
- Why avoid `async void`?

By the end of this chapter, Test Explorer in Visual Studio should look like the following screenshot:

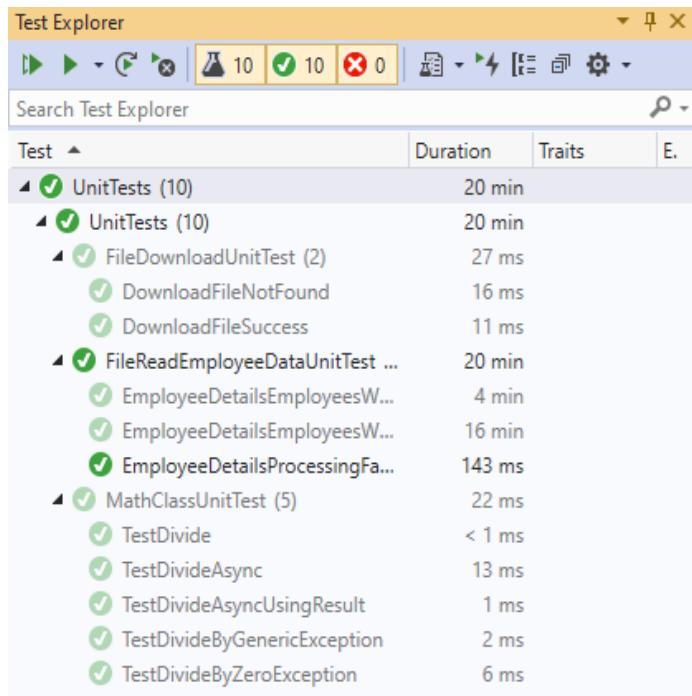


Figure 8.14: VS Test Explorer listing all unit tests

With the maturity in the unit testing framework and the support for asynchronous methods, developers can use samples in this chapter and write more robust unit tests.

In the next chapter, we will see tools and diagnostics in Visual Studio IDE to debug and troubleshoot issues in concurrent executions/parallel programming and multithreaded programs.

Exercise

1. Change return type of one of the asynchronous unit tests to `async void` and check the output? Does it always fail or works the same way, if yes, why?
2. What is the difference between MSTest, XUnit, NUnit?
3. Rewrite exception handling example using MSTest and NUnit?
4. Write a unit test with mock data for a web API that has asynchronous methods.
5. Write a unit test for an event handler that has an `async void` signature.

CHAPTER 9

Debugging and

Troubleshooting

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

~ Dijkstra

Once the development of the software is done, and the product is out for testing, the most dreadful word that any developer fears is a *bug*. **Bug** refers to any functionality in the product that is not working as it is intended to work. Bugs are inevitable in any software, more so in parallel and concurrent scenarios as they make the code complex and susceptible to human mistakes. Depending upon the impact a bug has on the business of your customer, a bug may be classified as high or low severity. High severity bugs may have rigid timelines to be met, and code fix must be shipped as soon as possible to lower the business impact. The developer needs to analyze the failing scenario and find the root cause of a bug so that it can be addressed quickly. The process of investigating the root cause or stepping through the code and validating if it's working as intended is referred to as debugging. In this chapter, we will discuss various tools and techniques to debug the multithreaded code in the world of .NET Core. Though Visual Studio Code is an excellent cross-platform editor, we will primarily focus on Visual Studio 2019 tools for debugging.

Structure

We will cover the following topics in this chapter:

- Debugging primer with Visual Studio 2019
- Profiling a multithreaded application
- Memory dumps – Collection and Analysis
- Identify and troubleshoot high CPU issues due to wrong multithreaded code
- Summary
- Exercise

Objectives

By the end of this chapter, the reader should be able to:

- Confidently debug multithreaded code using Visual Studio 2019 and its tools and extensions
- Profile multithreaded applications
- Identify the cause of High CPU issues that may arise due to wrong multithreaded code
- Learn to collect and analyze managed memory dumps using various tools

In the usual application development life cycle, a developer codes a component tests it, and sends it for testing. A good developer ensures that nothing untested leaves his desk. So, in the process of development, he executes the component multiple times and checks if the code flow and conditions are working as per his logic and expectation. Visual Studio 2019 provides a variety of tools to help developers debug, troubleshoot, and profile the code. In the next section, we would take a quick tour around the debugging and tooling support provided by Visual Studio 2019.

Note: All the screenshots and functionalities described in this chapter are from Visual Studio 2019 Enterprise edition. The other variants of Visual Studio 2019 may not have all the features. Also, depending upon the selected workloads, few items for Python, unity, or other languages may not be available in your Visual Studio 2019. These additional items are not used in the chapter or for parallel programming using .NET Core, so this should not be an issue.

Debugging primer with Visual Studio 2019

You may need to debug an application in design time or runtime, that is, while coding or while the code is deployed in a production environment. In the design time, you have the luxury of installing any tools that we may need to debug the application, but the same cannot be the case in the production environment as it may be running as

a sandbox environment or you may not be permitted to install software, to maintain the sanctity of the environment. In such cases, the best way to start debugging or troubleshooting an application is to check the log file of the application. Now log will appear if the application writes log files, and hence it is essential to have a useful logging framework in your application and log the appropriate details. The current trend is to make use of structured logging, that is, logs that can be queried. Microsoft Azure Application Insights is one of the splendid examples of logs that can be queried. Serilog, log4Net, nlog are popular logging framework libraries that provide different log level verbosity to log trace, information, debug and error details, and work well with .NET Core 3.1. This investment pays rich dividends later and reduces the time to identify issues.

C#.NET also has rich support for logging. These constructs reside in `System.Diagnostics` namespace. `Debug`, `Trace`, `Stopwatch` are few of the frequently used types to log details to investigate and troubleshoot the problems. In this section, we will discuss the debugging tools provided by Visual Studio 2019.

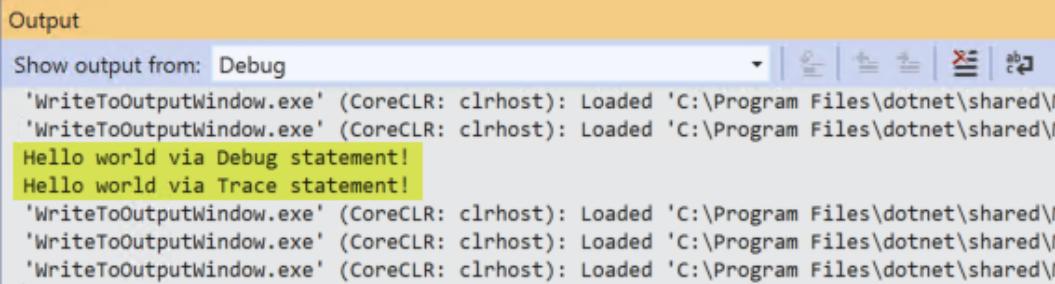
Visual Studio has a great toolbar and set of windows to help developers in their development, debugging, and deployment journey. Few of the most critical windows that are frequently used while debugging is:

- **Output window** (`Ctrl + Alt + O`) or in the top menu bar, **View | Output**. It displays all the status messages from the code or the Visual Studio IDE. When we use the `Debug` or `Trace` APIs in the code and debug the application, the messages logged via these APIs are displayed in the output window.

The following code would log the statements in the output window:

```
Debug.WriteLine($"Hello world via Debug statement!");
Trace.WriteLine($"Hello world via Trace statement!");
```

It can be seen in the following screenshot from the **Output** window:



The screenshot shows the Visual Studio Output window with the title bar 'Output'. The window displays log entries from the 'Debug' output source. The entries are:

```
'WriteToOutputWindow.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\...
'WriteToOutputWindow.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\...
Hello world via Debug statement!
Hello world via Trace statement!
'WriteToOutputWindow.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\...
'WriteToOutputWindow.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\...
'WriteToOutputWindow.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\...
```

The last two lines, 'Hello world via Debug statement!' and 'Hello world via Trace statement!', are highlighted with a yellow background.

Figure 9.1: Output window

- **Immediate Window** (`Ctrl + Alt + I`) or in the top menu bar, **Debug | Windows | Immediate**. This window is probably the most useful and most frequently used window while debugging and can be used to debug, evaluate

expressions, execute statements, or see the value of a variable. IntelliSense is very well supported by this window, so it's easier to write code statements in the next window. To execute, write the regular code and press enter. To see the value of a variable or method execution, prefix the code with a question mark (?) as shown in the next screenshot:

The screenshot shows the Immediate Window in Visual Studio. The title bar says "Immediate Window". The window contains the following text:
?anotherNumber
3
?anotherNumber
3
?aNumber
20
?aNumber+anotherNumber
23

Figure 9.2: Immediate Window

To switch from Command window to Immediate Window, type the command:

>immed

While, to switch from Immediate Window to Command window, type:

>cmd

Note the use of greater than (>) symbol before the command.

- **Watch** window ($Ctrl + Alt + W > 1$) or in the top menu bar, **Debug | Windows | Watch | Watch 1/Watch 2/Watch 3/ Watch 4** – there are four **Watch** windows in Visual Studio 2019, namely **Watch 1**, **Watch 2**, **Watch 3** and **Watch 4**. So, you can use 1,2,3 or 4 as the second command in conjunction with ($Ctrl + Alt + W$) to open the respective **Watch** window.

As the name makes it clear, these windows are used to watch variables and expressions while debugging. These windows are available only when debugging is in progress. We can watch multiple variables and expressions in the watch window. Similarly, there are four **Parallel Watch** windows, as well. These windows can be used to watch the expression/variable values which are executed on multiple threads:

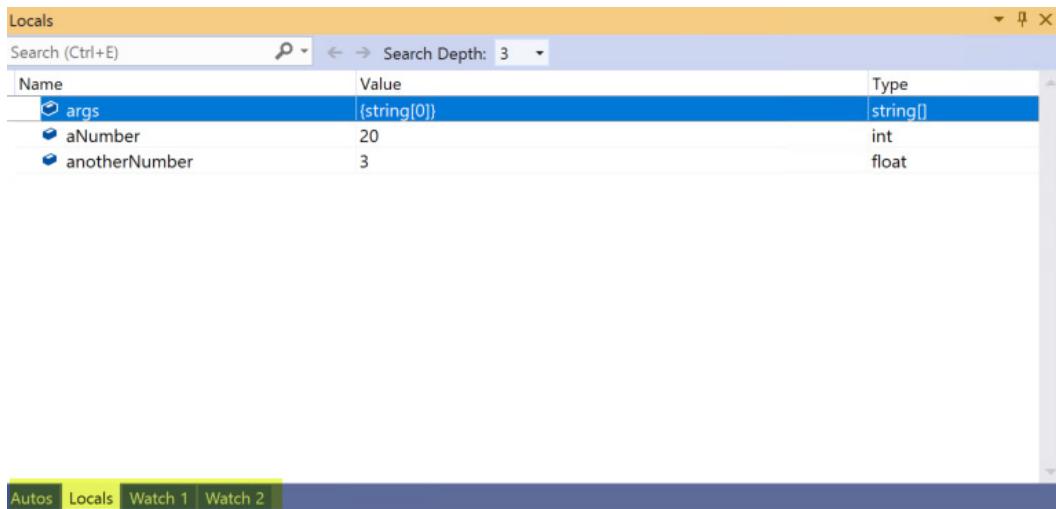


Figure 9.3: Autos, Locals and Watch window

- **QuickWatch window (*Shift + F9*)**: There is another window like the **Watch** window, called **QuickWatch** window, which can be used to watch a single variable/expression at the time of debugging. There are other windows like **Autos**, **Locals**, and so no, which displays the variables around the current breakpoint and variables in scope, respectively. We will not go into details of these as they are fundamental to debugging, and most developers would already be aware of it directly or indirectly:

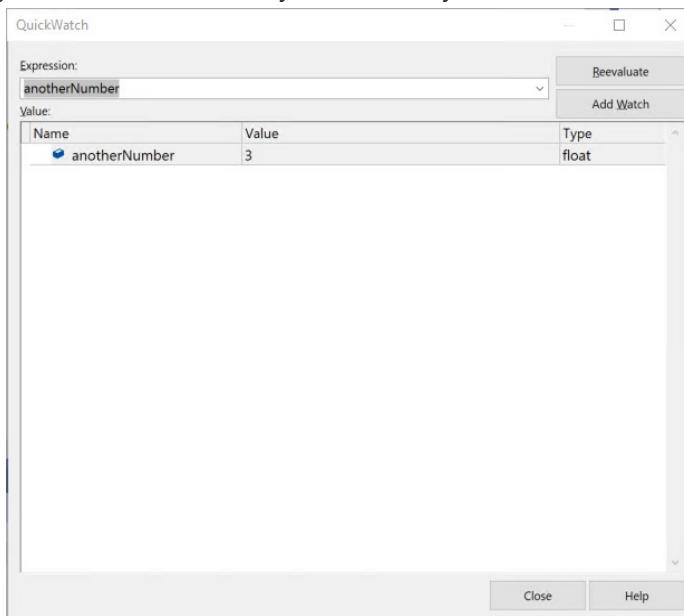


Figure 9.4: QuickWatch Window

In the **Solution Explorer**, you get the option to debug the selected project by right-clicking on the project. It displays a context menu. This menu has **Debug** as one of the context menu items, as shown in the next screenshot:

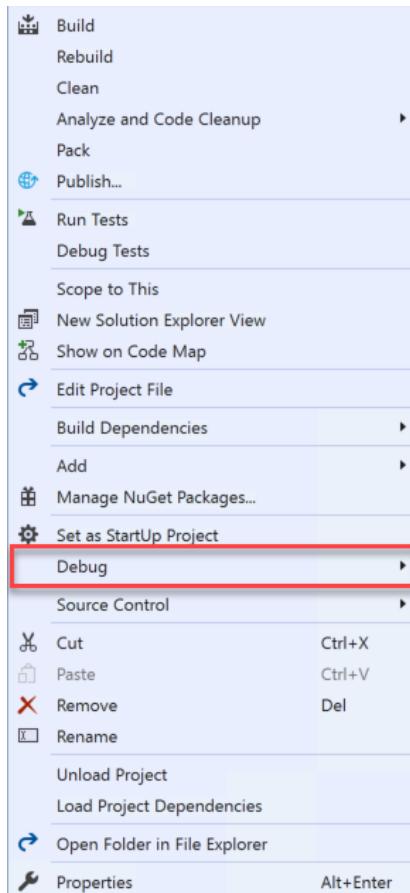


Figure 9.5: Right-click context menu

You can choose either **Start new instance** or **Step Into new instance** depending upon what you intend to do:

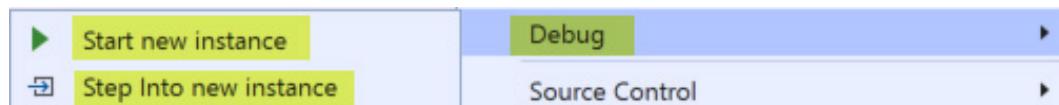


Figure 9.6: Start new instance and Step Intonew instance

- o **Start New instance:** Attaches the debugger to the project but continues the execution of the program until it encounters a breakpoint, exception, or completes till the end.

- o **Step Into new instance:** Attaches the debugger to the project and breaks the execution of the code in the entry point, so that the developer can then step through the code line by line to see and debug.

The top menu bar of the Visual Studio has a menu item called **Debug**, which groups and displays all the debugging tools that Visual Studio has to offer:

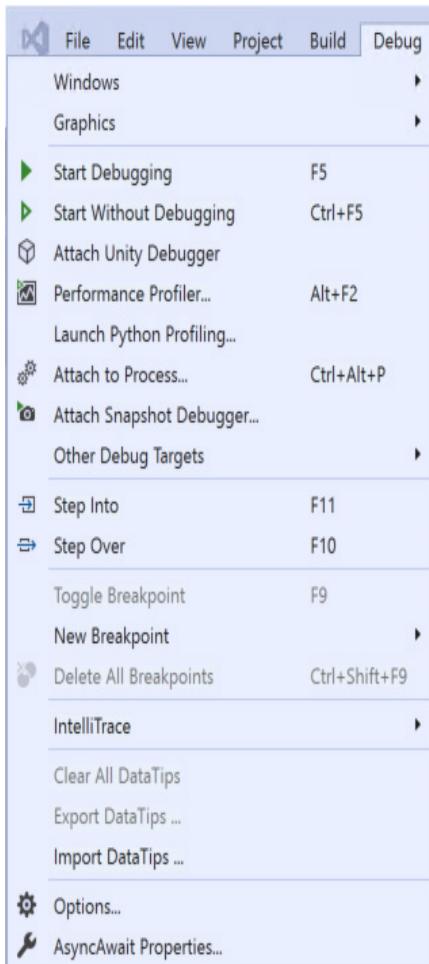


Figure 9.7: Debug menu

We see multiple items in the **Debug** menu. Few of these have child items as well.

These items in menu changes dynamically depending upon if the project is running or not from Visual Studio IDE as shown in the following screenshot:

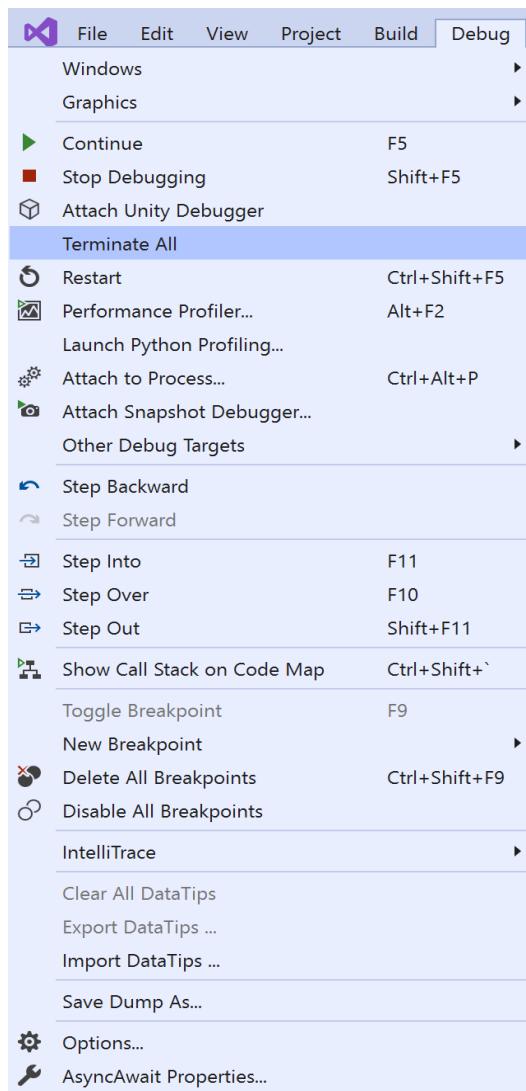


Figure 9.8: Debug menu with Debugger in a broken state

We will first discuss the purpose and usage of items that don't have nested child items and then move to items that have children.

Start Debugging (F5): Attaches the debugger to the selected project(s) and launches the project(s). Everyone who has ever done development with Visual Studio (any version) is familiar with this. While the debugger is paused on a breakpoint:

- *F5* is used to continue the execution
- *Shift + F5* can be used to stop the debugging
- *Ctrl + Shift + F5* to restart the application with debugging

The last two commands can be used even when the debugger is not paused, but the application is in running state.

Start Without Debugging (*Ctrl + F5*): Launches the selected project(s) without attaching the debugger. When you run an application or project without debugging, the debugger is not attached, but Visual Studio provides a way to attach debugger later in the flow by **Attach to Process**.

Attach to Process (*Ctrl + Alt + P*): There are times when we need to debug a process that is not running from the Visual Studio IDE or when the process is run from Visual Studio IDE, but the debugger is not attached to the process. In such cases, we can use **Attach to Process**, to select the process or processes to which debugger needs to be attached. It is convenient in debugging scenarios, which takes time to reproduce. Clicking on **Attach to Process** opens a dialog, as shown in the next figure.

Attach Snapshot Debugger: This can be used to debug the snapshots collected from live ASP.NET (.NET Framework 4.5 or above) or .NET Core (2.0 or above) web application on the event of an exception. It needs to be enabled and configured in your application first. Azure Application Insights monitors telemetry from the web app and collects snapshots from the exception throwing code in your application. It helps to debug and diagnose issues from production in on-prem servers or services deployed in Azure.

Step Into (*F11*): This helps to step into the code, one line at a time. If a method / property invocation is encountered in that line of code, then *F11* will step into the code of that method / property. We can use *Shift + F11* to step out of method / property if we do not wish to step over each line in it.

Step Over (*F10*): This is the same as **Step Into**, except that when it finds another method / property invocation in the line of code, it will not step into the method /

property code. The method / property code would execute, and the debugger will be brought to the next statement in the code flow:

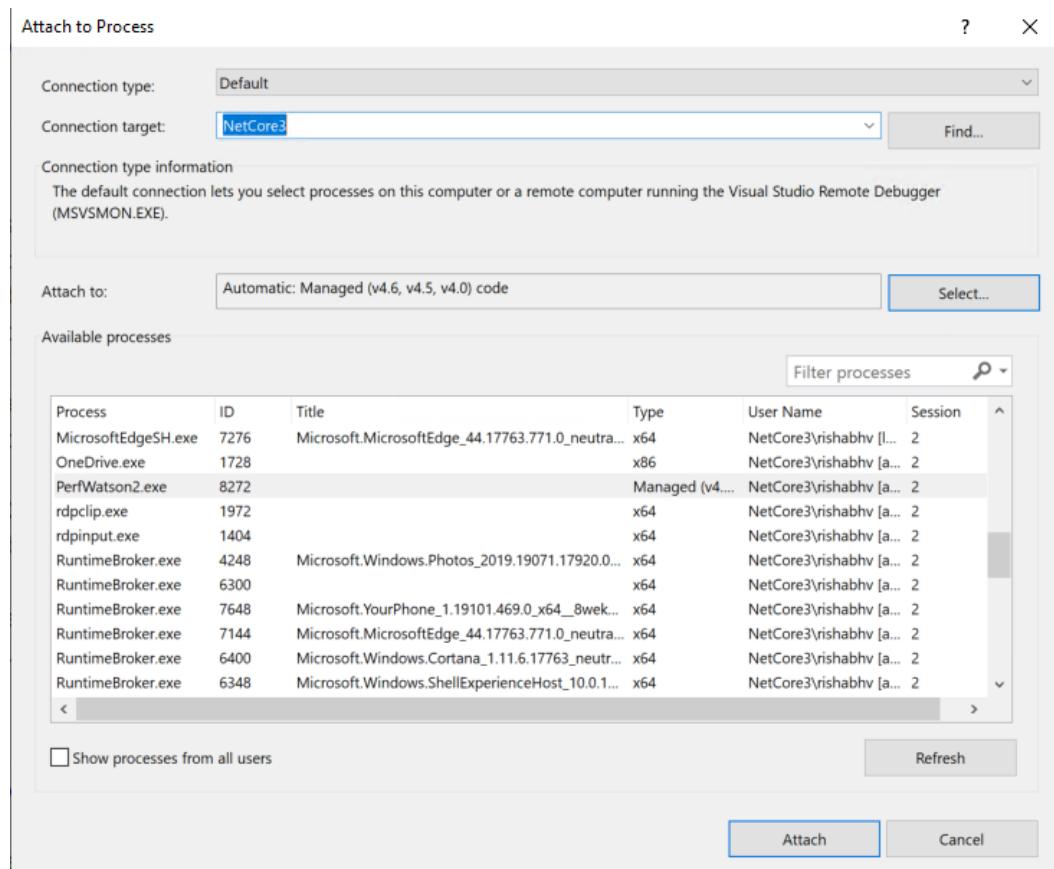


Figure 9.9: Attach to process

Depending upon if the process to be debugged is displayed or not in the dialog, you may want to check the checkbox for **Show processes from all users**. Even after checking this checkbox, if the process doesn't show up, and you are sure that the process is running, then you can click on the **Refresh** button to refresh the list of processes. Once you find the process to be debugged, select them and click the **Attach** button. The debugger would be attached to the selected process. For debugging managed .NET /.NET Core code, you would not need to change any of the values specified in **Connection Type** and target dropdowns.

New Breakpoint/Toggle Breakpoint (F9): This is used to insert a breakpoint in the code. You can either press F9, which is to toggle a breakpoint, but if a breakpoint doesn't exist, F9 will insert a breakpoint in that line of code. You insert breakpoint is that line of code where you want the debugger to pause, and you can either check the variables, parameters, or the control flow as needed.

There is support for conditional breakpoints as well. When you click on **Debug | New Breakpoint | New Function Breakpoint**, it provides a dialog to enter the **Function name, Conditions** when a breakpoint should be hit and Actions to be taken when a breakpoint is hit. You can also insert a breakpoint by clicking on the small left panel visible just before the code editor. On right-clicking a breakpoint, you can enter the conditions for the breakpoint to be hit and actions to be taken when that breakpoint is hit. It is fresh and handy in debugging:

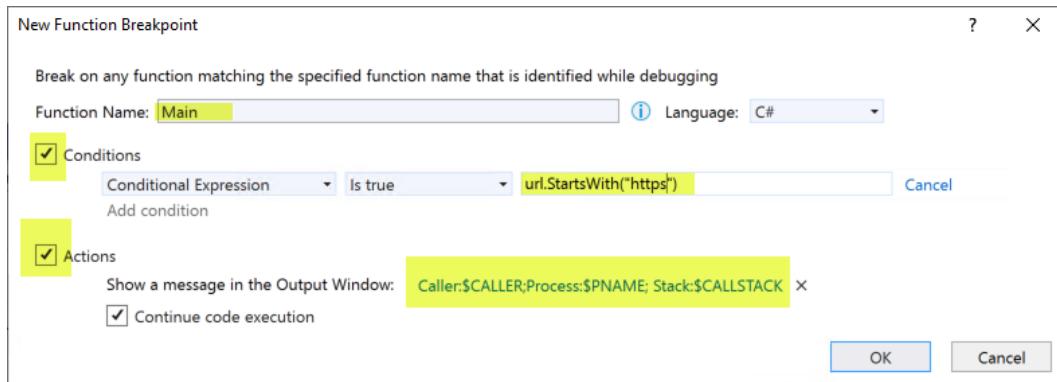


Figure 9.10: Breakpoints

In the previous screenshot, we see that we can specify a function name. Do click on the Information icon beside the function name to read its documentation.

Following the function name, we have checkboxes for conditions and actions. In the conditions, we can specify multiple conditions. The breakpoint is hit when all the conditions are met.

Then there is a checkbox for action, where we can specify a message to be displayed in the output window. It can be a plain text message or can make use of curly braces ({}) to log a variable value. Few predefined keywords can be used. Again, the information icon beside the textbox has the documentation and help information for using these keywords. While typing in the text box, you will get IntelliSense for these predefined values. When the output is logged, the values for these keywords are displayed. Apart from function breakpoints, we have the newly introduced data breakpoints as well. Earlier data breakpoints were available exclusively for C++ but starting .NET Core 3.1 and Visual Studio 2019 preview 2, and later, it is available for C# as well. Data breakpoint, as the name states, can be used to break the debugger when the data (or value) of a property or variable changes. To set a data breakpoint is natural and can be set from the **Watch** window. Just right click on the property or variable which you wish to monitor for data change and then click on **Break When**

Value Changes; this will insert data breakpoint on the property/variable and the debugger would break, whenever the value of this property/variable changes:

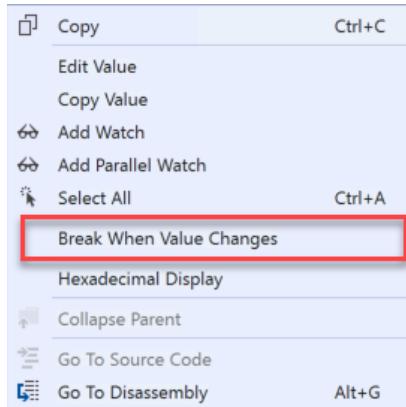


Figure 9.11: Data breakpoint

Tracepoint: Tracepoint is a breakpoint that has action defined to log the message to the output window. Tracepoints can be used to debug the applications which don't have detailed logging enabled, but we can still write the logs on the output window

Delete All Breakpoints (Ctrl+Shift+F9): Deletes all the breakpoints.

Disable All Breakpoints: Disables all the breakpoints. They can be enabled later if needed, so this differentiates them by deleting all breakpoints.

Save dump as: This is useful to save the memory dump file of the application. It collects a minidump of the memory of the process being debugged. The memory contains the objects created by the application, variables and their values, exceptions if any, thread stack traces, and so on, which is invaluable for debugging. These things can help us identify issues/exceptions in the application. We will troubleshoot a memory dump collected at the time of issue and identify the cause of an issue, a little later in this chapter.

Show Call Stack on Code Map: This can be used to display the call stack on the code map and can be useful to visualize the call stack in large, complex, and deep nested applications. A simple illustration is shown in the next screenshot:

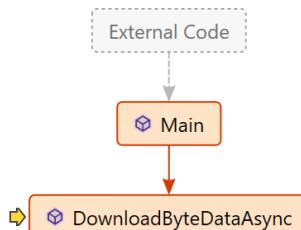


Figure 9.12: Code Map

Performance profiler: To perform the performance profiling of application. Profiling application helps us identify potential performance issues upfront, and we can fix them before shipping the application. We will do performance profiling of a multithreaded application, later in this chapter.

Options: To see and change the debugging options. The options are easy to understand and configure. The default settings in the options are good enough for us to debug. I would highly encourage the readers to get themselves familiarized with these options to use them effectively.

Below is a screenshot of the same.

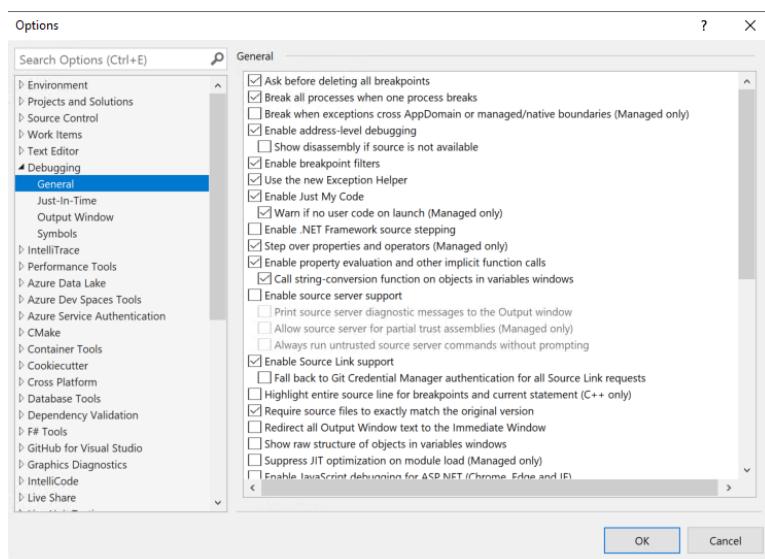


Figure 9.13: Debugging options

To make the debugging experience better and seamless, Visual Studio provides data tips that make it easier to visualize the values in the code editor itself at the time of debugging. The next image shows the DataTips for `aNumber` and `anotherNumber` in the code editor. You can also provide labels to them to distinguish and know them in the flurry of DataTips:

The code editor displays the following C# code:

```

Console.WriteLine("Hello World!");

// Immediate window
int aNumber = 100 / 5;
float anotherNumber = 200 / 66;
// See intellisense and values in immediate window.

Console.ReadLine();
}

```

A yellow callout box highlights the variable `aNumber` in the assignment statement. The callout box contains the label "aNumber" with the value "20" and the status "Initialized Value". Another yellow callout box highlights the variable `anotherNumber` in the assignment statement. This callout box contains the label "anotherNumber" with the value "0" and the status "Before initialized value".

Figure 9.14: DataTips

Apart from these, there are a few more menu items, which have nested children. For the sake of completion, let us quickly discuss these items as well. They are:

- **Graphics:** This can be used for graphics debugging.
- **Other Debug Targets:** To debug installed Windows Store or Universal Store Apps.
- **IntelliTrace:** To collect exciting IntelliTrace events. It can be used for time travel debugging for historical data back in time. The IntelliTrace settings can be adjusted to collect events, exceptions, CPU, and memory usage, which can be used to investigate issues. The saved iTrace file can be used to identify exceptions, CPU, and memory issues.
- **Windows:** This menu item has several windows that can be used for debugging, as shown in the next screenshot:

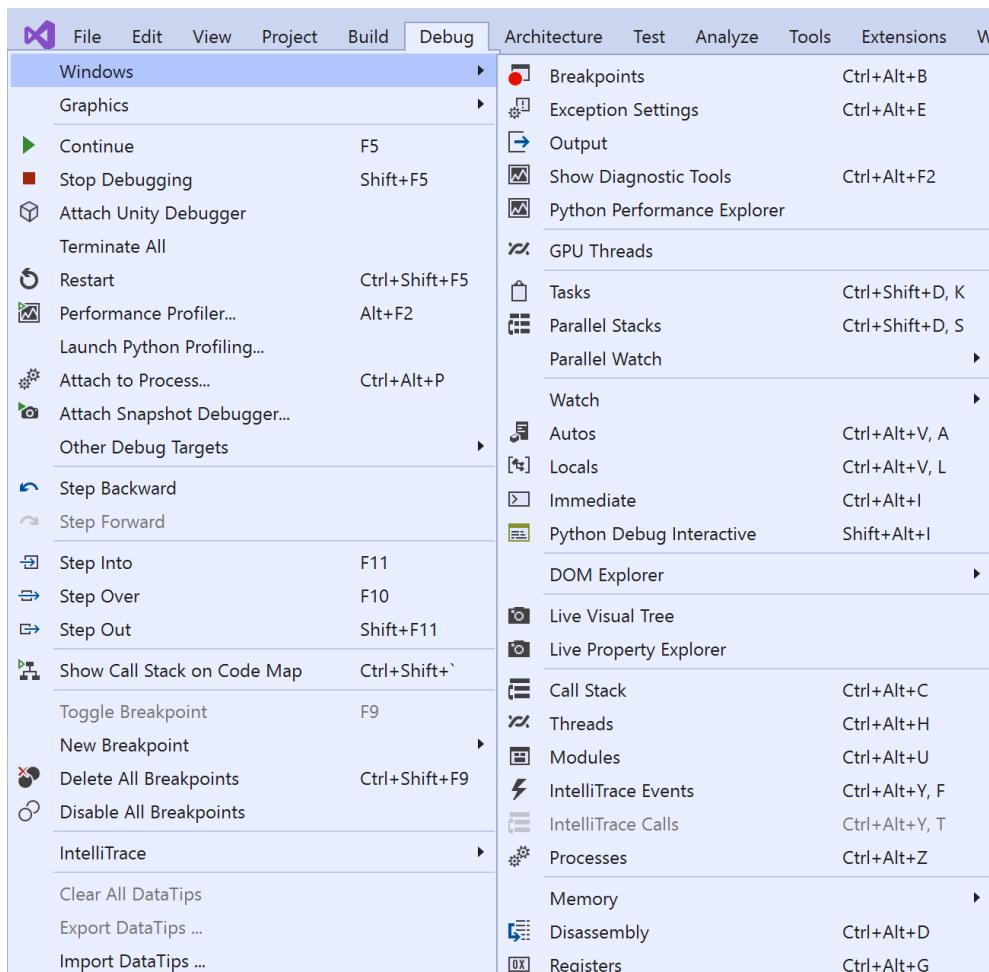


Figure 9.15: Debug windows

We have already seen and discussed, **Output**, **Watch**, **Parallel Watch**, **Autos**, **Locals**, **Immediate** windows. There are a few more important ones that we will discuss:

- **Breakpoints:** Lists all the breakpoints. You can enable / disable breakpoints from this window.
- **Exception Settings:** To change the exception settings. There are times when we do not want the debugger to break when an exception is thrown or vice versa. To change this behavior of debugger, we can modify the exception settings and check / uncheck appropriate exceptions. This window is a break when thrown. So, the debugger would break when a checked exception is thrown.
- **Show Diagnostic Tools:** To display the diagnostic tool, which will show the historical diagnostic data like CPU usage, memory usage, exceptions, and IntelliTrace events.
- **Tasks:** Displays the `System.Threading.Tasks.Task` objects which can run concurrently.
- **Call Stack:** Displays the method calls that are currently on the stack. The order in which methods are being invoked can be understood from the call stack, so to understand the execution flow of your application, the call stack window may be a handy tool.
- **Threads:** Displays the `System.Threading.Thread` running in your application code and see what threads are working when the breakpoint is hit.
- **Modules:** Displays the loaded assemblies and modules.

With fundamental debugging tools of Visual Studio 2019 in place, we can now proceed with some serious debugging and troubleshooting tasks. One of the most potent and essential activities during the development of the multithreaded application is to profile the application, which helps us discover performance and memory issues and avoid unnecessary issues from being reported when you ship your application. In the next section, we will discuss how we can leverage Visual Studio 2019 to profile your multithreaded application.

Profiling

To demonstrate profiling, we create a simple WPF app using .NET Core 3.1. The app takes URL as an input, and upon the click of the **Download** button, tries to download

the data from the URL and display the downloaded text in the bottom section of WPF UI. The screenshot of the WPF app is:

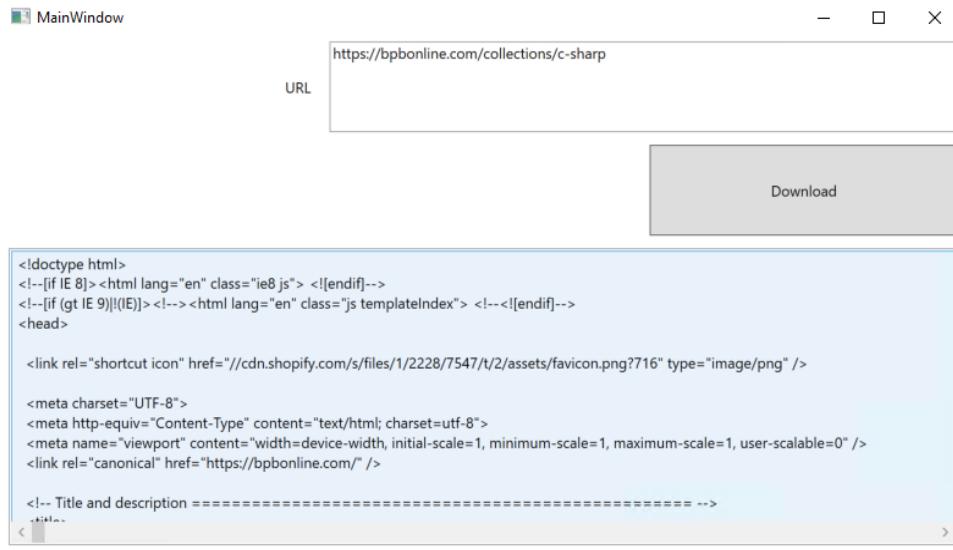


Figure 9.16: WPF App for profiling

The code of the Download button click event is:

```

private void BtnDownload_Click(object sender, RoutedEventArgs e)
{
    try
    {
        httpClient.BaseAddress = new Uri(this.txtUrl.Text);
        httpClient.DefaultRequestHeaders.Add("user-agent",
        "Mozilla/5.0 (Windows NT 10.0; WOW64)");
        var result = httpClient.GetAsync("/").Result;
        result.EnsureSuccessStatusCode();
        var output = result.Content.ReadAsStringAsync().Result;
        this.lstData.Items.Add(output);
        this.lstData.UpdateLayout();
    }
    catch (Exception ex)
    {
        this.lstData.Items.Add(ex.ToString());
    }
}

```

It is observed by the developer that after clicking the download button, while the application is still trying to download the content from the URL, the UI of the application is frozen and doesn't accept user keyboard or mouse input. We are supposed to profile the application and find out why that is. Though with the knowledge accumulated thus far, we can just review the above code and recommend using `async await`, we will profile this application using Visual Studio 2019 profiling tool and find out the issue.

To profile this application, we will follow the following steps:

1. Navigate to **Debug | Performance Profiler** or (**Alt + F2**).
2. It will launch the performance profiler window, and you would need to check the checkboxes to configure the profiler. Since we are experiencing slowness, we will check **CPU Usage** and **Application Timeline** checkboxes as highlighted in the next screenshot:

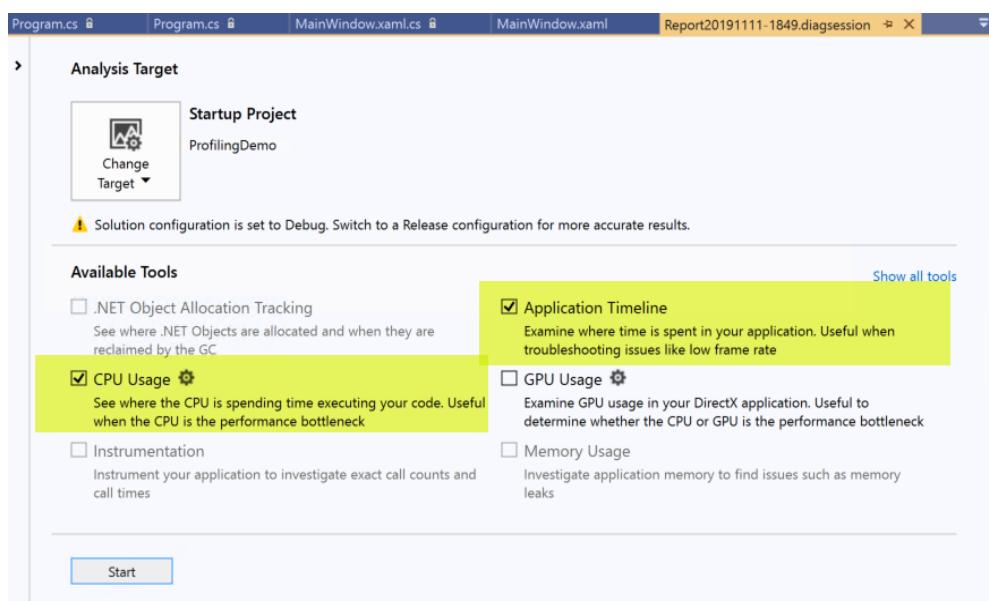


Figure 9.17: Performance Profiling

3. The **Analysis Target** is already set to our project that we want to profile, so click on the **Start** button at the bottom. It would launch our WPF application.
4. Now, the slowness was observed when the **Download** button was clicked, so we want to profile this scenario. Click on the **Download** button in the application and wait for it to complete.
5. Once the download operation is completed in the WPF app, click on stop collection in the profiler, to stop the profiling of our WPF app. It will generate an excellent graphic output, which can be used for our investigation. Since

we selected **CPU Usage** and **Application Timeline** analysis, we will get the breakup of how much time was spent in doing what? What took most CPU cycles? What was UI thread blocked on?

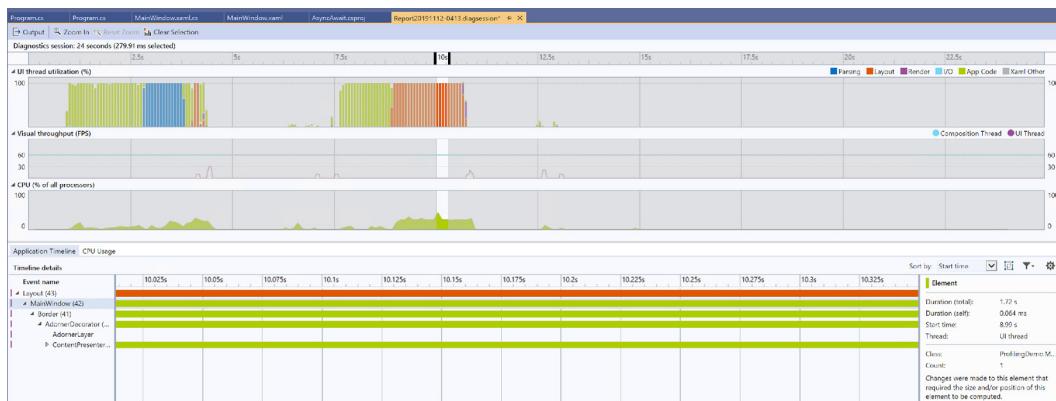


Figure 9.18: Performance Profiling

In this profiling report, we can see that the UI thread is 100% utilized at start-up and when the **Download** button is clicked. We can also see what the activity that contributed to thread utilization was. In the start-up, layout contributed to the utilization, while on the download button click, app code contributed most to the UI thread utilization. You can click on the individual item at the time of interest in the report to dive inside and get more details. Since we wanted to know about the issue causing the app to be unresponsive, we find out that UI thread is unresponsive as it is 100% utilized by the app code and layout rendering after the click of the **Download** button. To fix this, we need to free up UI thread from the massive operation. It can be done by offloading the download operation to a background thread or by using `async await`.

The better version of code which fixes this issue would be:

```
private async Task GoodCodeAsync()
{
    try
    {
        httpClient.BaseAddress = new Uri(this.txtUrl.Text);
        httpClient.DefaultRequestHeaders.Add("user-agent",
        "Mozilla/5.0 (Windows NT 10.0; WOW64)");
        var result = await httpClient.GetAsync("/");
        result.EnsureSuccessStatusCode();
        var output = await result.Content.ReadAsStringAsync();
    }
}
```

```
        this.lstData.Items.Add(output);
    }
    catch (Exception ex)
    {
        this.lstData.Items.Add(ex.ToString());
    }
}
```

We will not always have the luxury to install and use Visual Studio in the server or machines in which the issue is encountered. There are times when the issue occurs in servers or other non-development machines, where due to customer policies, we may not be able to install Visual Studio or any other debugger. These issues may be high CPU usage, memory leak, or any exception. To debug such kinds of multithreaded applications, we can make use of memory dumps. We collect memory dumps from the faulting environment and ship it to the development team, which can analyze these dumps and find out the issues. Before we learn to collect, memory dumps, and analyzing it, we must first understand memory dumps and why they can be used to investigate such issues.

Memory Dumps

A memory dump contains the state of the process. The primary objective of generating the memory dump of the process is to investigate process failures or issues when live debugging using debuggers is not possible. Memory dump or just dump file represents the point in time state of the process, and depending upon the memory space taken up by the process may be a large file. This file can be shared with the development/engineering team to analyze the dump later and share the analysis. This debugging is also referred to as post-mortem debugging. What is contained in the dump file that helps the developer do the analysis? The dump file may contain the following:

- **Heap memory:** This has all the variables and objects that are created by the process.
- **Exception objects:** Though it need not be called out explicitly, heap memory also contains any exception objects that are created so that we can find out exceptions in the dump.
- **Call stack of all the executing threads:** At the time memory dump was collected, the function and method calls that are being made by all the threads are collected for all the threads. It can be used to find out what each thread was doing when the memory dump was collected.

- **Thread Environment Block:** This contains the state and Id of each thread, querying which developer can find out if the thread was in running, waiting, or any other state.
- **Module Information:** All modern software and applications have multiple files and assemblies that are loaded in the process memory space. This information, along with their version, is also collected as part of the memory dump.

As we can see, memory dumps contain a variety of information. Dump files are written by a tool, generally debugger. Based on the information collected, it can be categorized into two types:

- **Full memory dump:** This contains a copy of the contents of the entire virtual memory of the process. It is useful to discover and troubleshoot unknown issues. An engineer analyzing this dump can look up anywhere in the memory to locate objects, variables, exceptions, and even disassemble the code to diagnose the issue. The downside of this type of dump file is that since it contains all this detailed information, it can be significantly large depending upon memory being used by the process. The collection of this type of dump may take a lot of time and may impact the environment from functioning correctly at the time dump is being collected. The collection process virtually freezes the process to collect its memory snapshot.
- **Mini memory dump:** This is a miniature version of virtual memory of the process, compared to the corresponding full memory dump, and generally has lesser file size against the full memory dump collected at the same time for the same process. It is made possible by the debugger that writes the dump file. The debugger can configure the subset of information to be written from the entire virtual memory space of the process. It may have a disadvantage to an engineer analyzing the dump that the required information to and investigate the issue may be missing from the dumps.

Now that we have discussed memory dumps in some details, let us see the action. In the next section, we will learn to collect and analyze the memory dump using various tools.

Note: In today's world, where privacy is of utmost importance to everybody, we need to be cautious in what we collect from the production server and make the customer aware of it to avoid any non-compliance or privacy breaches.

Collecting memory dumps

Before we learn to collect memory dumps, we should first write a multithreaded application which has an issue, so that we can collect the memory dump for that application and investigate it. So, we will write the wrong multithreaded code to create an issue first. Below is the code from our multithreaded application.

This a simple .NET Core 3.1 console app. There are comments to help the reader understand the purpose of code, but we will quickly recap it below:

```
static void Main(string[] args)
{
    //// Look up data to be populated
    Dictionary<int, int> lookupData = new Dictionary<int,
int>();

    Console.WriteLine("This sample demonstrates bad code written
using multi-threading. We will use this sample learn to collect and
analyze dumps via different tools !");

    // Dumps are collected at a point of time. To make
demonstration easier, we will perform operations inside an infinite loop.

    while (true)
    {
        try
        {

            Parallel.For(0, 1000000, new ParallelOptions() {
MaxDegreeOfParallelism = Environment.ProcessorCount }, (i) =>
            {
                //// Process data returned from API and
added it to a collection.

                Thread.Sleep(20);
                lookupData.Add(i, GetValueFromAPI(i));
            });
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception occurred in processing
data {ex.ToString()}");
        }
    }

    static int GetValueFromAPI(int i)
{
```

```

    //// Simulate delay of 1sec for fetching data from DB/API
    Thread.Sleep(1000);
    var result = new Random().Next(0, i);
    return result;
}

```

1. The console app needs to populate lookup data having key and value in a dictionary. The key is an integer, and the corresponding value is fetched from an API call/DB.
2. The keys, as we can see from the code, start from 0 and goes all the way till 1000000.
3. To simulate fetching data for a given key from an API/DB, we have introduced a delay in the `GetValueFromApi` method, generated a random number, and returned as value.
4. Since the iteration is significant and would take a lot of time, so the developer decides to leverage the power of all the CPUs to execute the code faster and so used `Parallel.For` loop.
5. During unit testing and runs, the developer found out that the behavior of this console app is unpredictable. At times, the lookup data is populated, while at other times, the exception is encountered, and lookup data fails to populate.
6. Since the memory dump represents the point in time state of the process, to make the dump collection process more accessible, we have wrapped the code inside an infinite loop.

This sample is quite an easy one, and the problem can be easily identified by an excellent and detailed code review. However, we will use this sample to learn about memory dump collection and analysis using different tools. The name of the executable of this lousy code is `BadMultithreadedCode.exe`.

The next screenshot illustrates an occasional exception that may occur while debugging our wrong sample code in Visual Studio 2019:

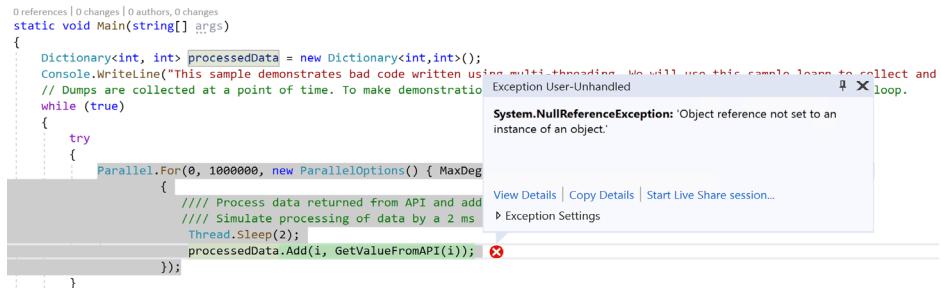


Figure 9.19: Exception in wrong multithreaded code

In the next section, we will see different tools to collect the memory dump of this process to learn different ways of collecting memory dumps:

Task Manager: The easiest way to collect memory dump is by using **Windows Task Manager**. It is installed by default in the Windows operating system. First, run the application. When it encounters an issue, like CPU or exceptions, navigate to the **Task Manager**. You can navigate to **Task Manager** by:

- Typing task manager in **Run** command (*Windows + R*)
- Right-clicking on the **Taskbar** and then clicking on **Task Manager** in the context menu
- Pressing *Ctrl + Shift + Esc*

In the **Task Manager**, click on the **Details** tab and identify the process for which we need to collect the memory dump. Right-click on the process and then click on **Create Dump File**. The dump file will be created, and the path to the dump file will be displayed in the dialog. Navigate to that path and copy the dump file. It is created as a minidump file:

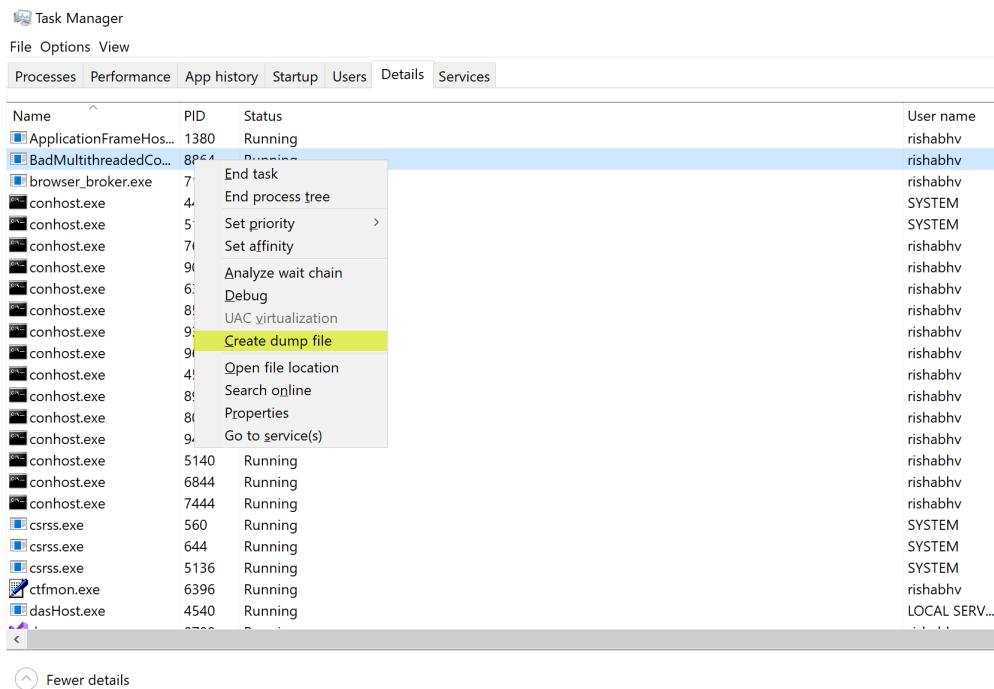


Figure 9.20: Create a dump file from Task Manager

ProcExp: A great tool from the Sysinternals Suite of tools is ProcExp or Process Explorer. I see it as an advanced version of the **Windows Task Manager**. It can be helpful to collect both full memory dump and mini memory dumps. It doesn't

need any installation and works by just copying a small exe (latest version is less than 2 MB at the time of writing this chapter) in your impacted machine. Like **Task Manager**, we can also attach a debugger to a process, by doing a right-click on the process and then click on **Debug**. To collect memory dump using ProcExp, we can follow these steps:

1. Launch ProcExp, by double-clicking on the `proexp.exe` in the downloaded location.
2. Find the process that is encountering issues. It is `BadMultithreadedCode.exe` in our sample scenario.
3. Right-click on the process and then click on **Create Dump** and then click on **Create Minidump** or **Create Full Dump** to collect mini dump or full dump depending upon your need.
4. Specify the dump file name and location where you want to save it, in the save dialog, and click on **Save**.

That's it! The memory dump file is successfully created:

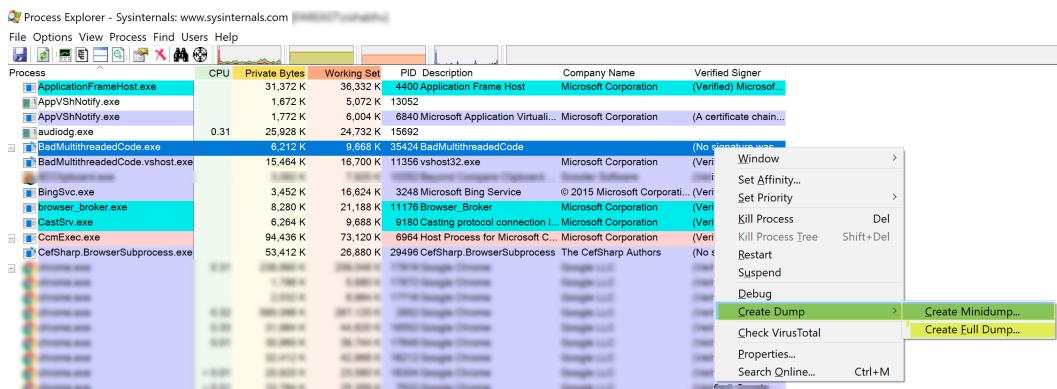


Figure 9.21: Create dump using ProcExp

ProcDump: ProcDump is a command-line utility that monitors the CPU spikes and collects the memory dumps for the crash, high CPU, or hang scenarios. It can also be used as a general-purpose utility to collect the memory dump. Since it is a command-line utility, it finds the first usage in automation scripts and other utilities that needs to collect memory dumps programmatically. Like ProcExp, it also comes from Sysinternals and doesn't need installation and works directly by copying the executable to the target machine. ProcDump can be used to collect memory dumps of a process based on the predefined threshold value. When the threshold is exceeded for the specified duration, the dump is automatically collected. Due to this cool feature and otherwise, this tool finds usage even in Microsoft Azure web apps, and you get this tool by default in the Kudu site / tools folder.

ProcDump being command line has several commands, and covering all the commands in detail needs a chapter in itself. If you need to use ProcDump, you automatically qualify as an intermediate or advanced user, and so you can very well go through the help of ProcDump to see and learn the commands of ProcDump. For a quick reference, a screenshot from the help command of ProcDump is shown in the next figure. I just launched command prompt, navigated to the location where `procdump.exe` was downloaded and typed `procdump.exe`, and clicked *Enter*. Then I took the screenshot of what followed. If you like to learn online, you can read the detailed documentation of ProcDump from this official Microsoft URL <https://docs.microsoft.com/en-us/sysinternals/downloads/procdump>:

```
D:\>Procdump>procdump.exe

ProcDump v9.0 - Sysinternals process dump utility
Copyright (C) 2009-2017 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com

Monitors a process and writes a dump file when the process exceeds the
specified criteria or has an exception.

Capture Usage:
  procdump.exe [-mm] [-ma] [-mp] [-mc Mask] [-md Callback_DLL] [-mk]
  [-n Count]
  [-s Seconds]
  [-c|-cl CPU_Usage [-u]]
  [-m|-ml Commit_Usage]
  [-p|-pl Counter_Threshold]
  [-h]
  [-e [1 [-g] [-b]]]
  [-l]
  [-t]
  [-f Include_Filter, ...]
  [-fx Exclude_Filter, ...]
  [-o]
  [-r [1..5] [-a]]
  [-wer]
  [-64]
  {
    {{[-w] Process_Name | Service_Name | PID} [Dump_File | Dump_Folder]}
  |
    {-x Dump_Folder Image_File [Argument, ...]}
  }

Install Usage:
  procdump.exe -i [Dump_Folder]
  [-mm] [-ma] [-mp] [-mc Mask] [-md Callback_DLL] [-mk]
  [-r]
  [-k]
  [-wer]

Uninstall Usage:
  procdump.exe -u

Options:
  -mm      Write a 'Mini' dump file. (default)
           Includes the Process, Thread, Module, Handle and Address Space info.
  -ma      Write a 'Full' dump file.
           Includes All the Image, Mapped and Private memory.
  -mp      Write a 'MiniPlus' dump file.
           Includes all Private memory and all Read/Write Image or Mapped memory.
           To minimize size, the largest Private memory area over 512MB is excluded.
           A memory area is defined as the sum of same-sized memory allocations.
           The dump is as detailed as a Full dump but 10%-75% the size.
           Note: CLR processes are dumped as Full (-ma) due to debugging limitations.
  -mc      Write a 'Custom' dump file.
           Include memory defined by the specified MINIDUMP_TYPE mask (Hex).
  -md      Write a 'Callback' dump file.
           Include memory defined by the MiniDumpWriteDump callback routine
           named MiniDumpCallbackRoutine of the specified DLL.
  -mk      Also write a 'Kernel' dump file.
           Includes the kernel stacks of the threads in the process.
           OS doesn't support a kernel dump (-mk) when using a clone (-r).
```

Figure 9.22: ProcDump help and usage

To collect a memory dump for our `BadMultithreadedCode.exe`, we can follow the given steps:

1. Launch the command prompt and navigate to the location where `procdump.exe` is downloaded.
2. Find the process ID of the process (`BadMultithreadedCode.exe`) for which memory dump needs to be collected from Task Manager/Process Explorer. The PID column value of the process is what we need. (You can find PID in the **Details** tab of Task Manager. The value under the PID column for your process is what you need)
3. Type the command `procdump -ma <PID>`.
4. The preceding command would write the memory dump of the process in a path and would display that path in the console, so you can use that path to find the location of a dump file. Following is the screenshot of how this looks like:

```
D:\Procdump>procdump -ma 35424

ProcDump v9.0 - Sysinternals process dump utility
Copyright (C) 2009-2017 Mark Russinovich and Andrew Richards
Sysinternals - www.sysinternals.com

[17:47:40] Dump 1 initiated: D:\Procdump\BadMultithreadedCode.exe_191115_174740.dmp
[17:47:40] Dump 1 writing: Estimated dump file size is 59 MB.
[17:47:40] Dump 1 complete: 59 MB written in 0.2 seconds
[17:47:40] Dump count reached.
```

Figure 9.23: ProcDump dump collection command

DebugDiag: DebugDiag is yet another tool that we can use to collect the memory dump. It comes from Microsoft. The tool makes use of `Microsoft.Diagnostics.Runtime.dll`, also referred to as `ClrMD`. DebugDiag installation comes with three executable components apart from samples and a detailed help/documentation file:

- **Collector:** To collect a process memory dump.
- **Rules Builder:** Provides a UI driven way to create new rules for DebugDiag. The output of this is a XAML file that can be used with the Analyzer for analyzing the dump file.
- **Analyzer:** To analyze the memory dumps with the selected analysis rules.

To collect the process memory dump using DebugDiag Collection tool:

1. Launch DebugDiag Collection tool. This will, by default, present you with a wizard-style dialog.
2. Select the **Rule Type**, that is, **Crash, Performance, or Native (non-.NET) Memory and Handle Leak**. If you want to investigate exceptions that maybe

crashing your app, use crash rule, or for performance-related issues like high CPU, slowness, slow responsiveness, and many more, use hang analysis. Native memory is generally not useful to help analyze managed (.NET/.NET Core) memory dump. The description lists the scenarios in which a rule type should be selected:

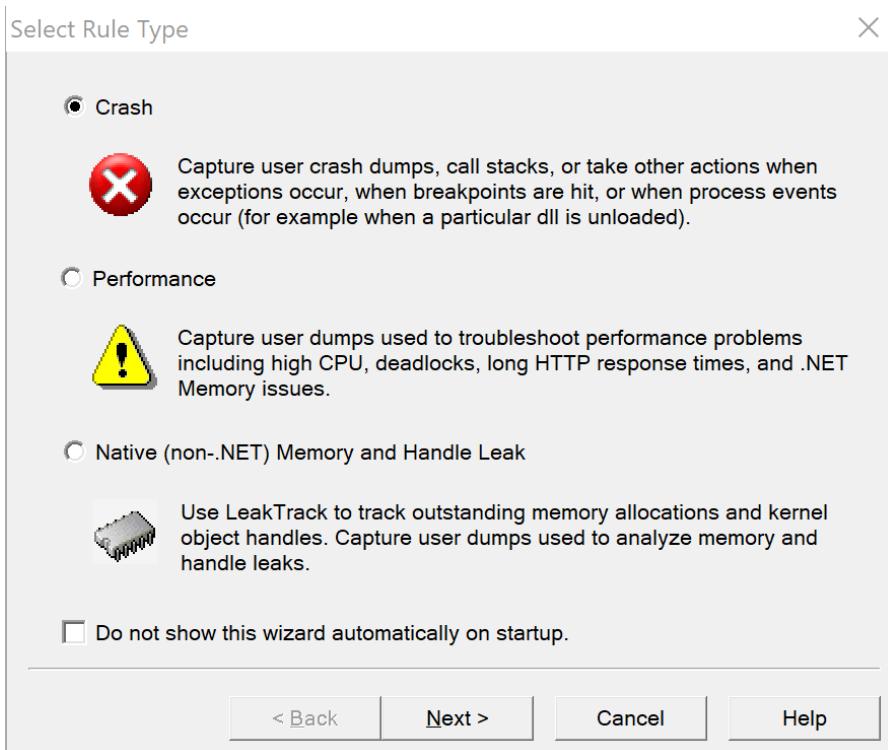


Figure 9.24: DebugDiag - Select Rule Types

3. Select the appropriate rule and then click on the **Next** button. Depending upon the chosen rule type, the next dialog would be different. Again, the description in the dialog would be good enough for you to make a selection. Select the appropriate rule and click **Next** again. For this demo, we shall select **Crash** in first dialog and A specific process in its successor (**Select target** type dialog), and then we would be presented with Select Target dialog, where we will select our sample **BadMultithreadedCode.exe**. Next few dialogs require optional inputs, so unless you know that the configuration

needs to be changed, just keep clicking **Next** in subsequent dialogs and finally check activate the rule now as shown in the next screenshot:

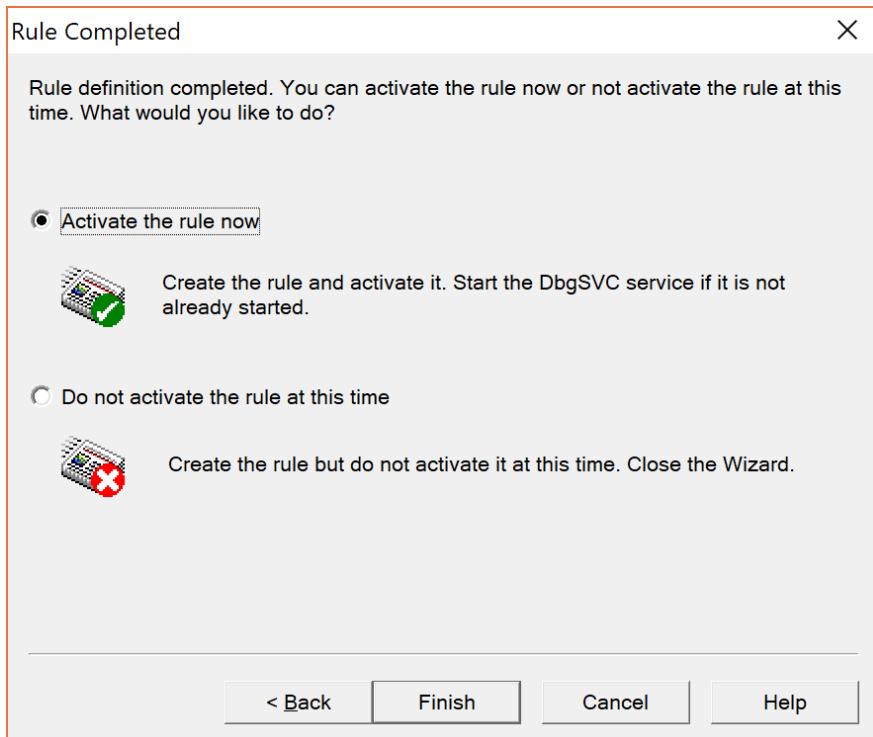


Figure 9.25: Rule Completed dialog

4. Whenever an exception occurs, the collection tool would start collecting the memory dumps and update the dump count and path in the user interface.
5. Note that once you are done with the collection, do not forget to deactivate the rule and stop the collection tool, else it would keep capturing dumps and continue taking space in your hard disk drive.

I would highly encourage the readers to have a glance at the documentation as it is the officially recommended guide to use the tool. It is a potent tool and can be used to collect memory dumps using predefined rules, like first chance exception (exception handled by try...catch block) or second chance exception (generally a crash). It also offers an interactive user interface, making it user friendly. It also has developer guidance to extend the DebugDiag by writing new analysis rules by implementing a simple interface using ClrMD.

dotnet-dump: So far, all the dump collection tools that we saw were cool, but they all work only with Windows. .NET Core is cross-platform, and hence we may need to collect the dumps in Non-Windows environment such as Linux as well. To this end, Microsoft has shipped a command-line tool called dotnet-dump that can be used to

collect and analyze memory dumps without having a native debugger installed in both Windows as well as Linux. Here, it's essential to call out that `dotnet-dump` is not supported on macOS. There is an extension for **WinDbg (Windows Debugger)** called **SOS (Son of Strike)** that is used to analyze the managed memory dumps. The same SOS commands can be used with the `dotnet-dump` command-line tool to analyze the crash and memory pressure related issues.

At the time of writing this chapter, this command-line tool doesn't get installed when installing Visual Studio 2019. It must be installed explicitly. It may change in the future with new updates.

To install the `dotnet-dump` tool:

1. Open the command prompt.
2. Type `dotnet tool install -g dotnet-dump` and press *Enter*.

It will install the `dotnet-dump` tool which supports commands for memory dump collection and analysis:

```
C:\Users\rishabhv>dotnet tool install -g dotnet-dump
Since you just installed the .NET Core SDK, you will need to reopen the Command Prompt window before running the tool you installed.
You can invoke the tool using the following command: dotnet-dump
Tool 'dotnet-dump' (version '3.0.52901') was successfully installed.
```

Figure 9.26: Install dotnet-dump

Being a command-line tool, the tool supports several commands. To familiarize yourself with the exhaustive list of commands, I would recommend the official Microsoft documentation for this utility at <https://docs.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-dump>.

To collect the dump using `dotnet-dump`, we need to launch the `dotnet-dump` tool and type the following command:

```
dotnet-dump collect -p <PID> --type heap --diag
```

The same is shown below:

```
C:\Users\rishabhv>dotnet-dump collect -p 6788 --type heap --diag
Writing minidump with heap to C:\Users\rishabhv\dump_20191116_093402.dmp
Complete
```

Figure 9.27: dotnet-dump collection command

It completes our quest to explore tools to collect the memory dumps. There are numerous other tools and scripts which collect the memory dump as well, but these are the most used and user-friendly tools I came across. In the next section, we will explore the tools and techniques to analyze the collected memory dumps and find the root cause of the issue.

Analyzing memory dumps

Collecting memory dump is just a first step in troubleshooting. Actual troubleshooting happens while analyzing the collected dumps. There are numerous tools to analyze memory dumps. In this section, various tools and techniques to explore what was going on in the process when the memory dump was collected.

Important Notes:

1. The bitness of debugger is essential to analyze the dump file. x64 version of debugger should be used to analyze the memory dump of the x64 process. Else, you may run into issues.
2. When the dump files are zipped and uploaded to a shared location on the cloud-like OneDrive or SharePoint sites, so that an engineer/development team can access it, it may get blocked, as a security measure so that it doesn't execute upon downloading. So, before opening the dump file, it is always a good and recommended practice to right-click on the file and check in the Properties of the file, it is not blocked. In case the file is blocked, you will get an additional checkbox (**Unblock**), which you should check and unblock the file.
3. For all production and enterprise applications, memory dumps must always be transferred securely using a storage account, which makes use of authentication as dumps may contain confidential or privileged information.
4. Memory dump of a process contains the entire working set of the process, so it may also contain personal information and other sensitive personally identifiable information. Passwords, SSN, Credit Card number, and so on, to name a few. With the right tools and commands, these can be retrieved and misused, so do ensure you keep the privacy and security of your customer and company at the top, before collecting or analyzing the dumps.

Once the dump reaches the developer's machine, he can make use of any tool that he has at his disposal to visualize and analyze what the memory dump contains. Visual Studio being the preferred development and debugging tool of choice, we will first see how we can use Visual Studio 2019 to analyze the collected dump file.

Visual Studio: Visual Studio has excellent support to visualize and investigate memory dumps. Like with most Microsoft products, the steps to analyze a memory dump using Visual Studio is easy. To do so, please use the following steps:

1. Open the dump file in Visual Studio by right-clicking on the dump file and then Open with Visual Studio 2019. Alternatively, you can open the dump directly in Visual Studio by **File | Open | File** (*Ctrl + O*) and select the dump file.

- Visual Studio will open the dump file and display the summary view of the dump file as shown:

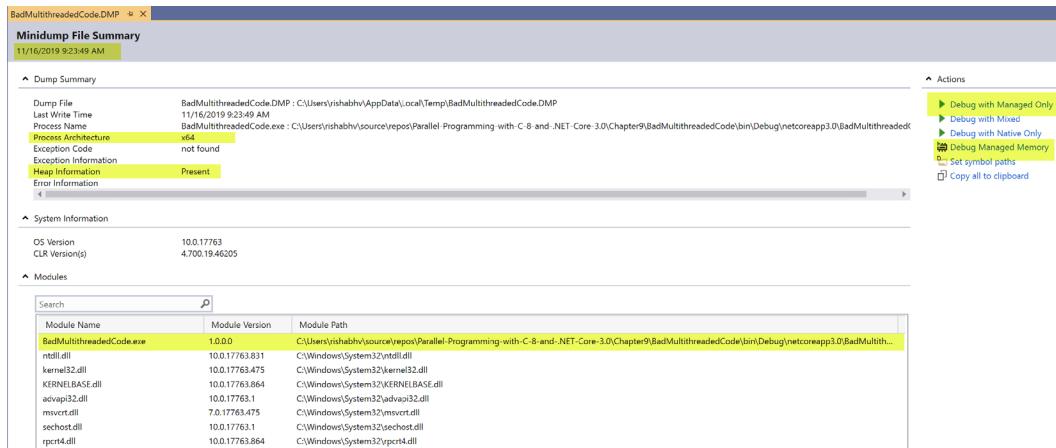


Figure 9.28: Dump file opened in Visual Studio

We can see:

- The dump file name, location.
 - Time, when memory dump was created.
 - Whether heap information is present or not.
 - The process architecture, that is, whether the process was x86 or x64.
 - The CLR and OS version.
 - The module name, version, and path loaded in the process.
 - On the right-hand side, we see the list of available actions, which are enabled based on the heap information being present or not. I would encourage the readers to explore these actions. The symbol path is essential, and that is from where debugger loads symbol or PDB files and knows about the source code line number mapping while debugging, among other debugging information. It should be set to use the Microsoft symbol server and NuGet symbol server paths to have the NuGet dll and Microsoft symbols downloaded and used from the local machine. Please note that this is a massive operation to download symbols and may take a while of time as well as hard disk space for the first time use.
- For this demonstration, we would click on **Debug Managed Memory**. This option will be enabled if heap information is present in the dump.
 - It would display the Heap View of the managed memory and offers the following cool functionalities that can be used to identify the memory issues like:

- **Memory leaks:** This can be done by comparing one memory dump to another memory dump of the same process collected some time apart to identify what all objects were not cleaned up by garbage collection and causing the surge in memory. Note memory leak can be confirmed only after analyzing two or more dumps, comparing them, and observing the memory trend of the process.
- Identify how many instances of types are there in the memory and what are the references and path to the roots of those objects. It can be done by clicking on an object type of interest and noticing their Referenced Types or Path to Root in the bottom section.
- See the memory space occupied by an instance of object type and including all objects referenced by them:



Figure 9.29: Heap View

5. At this point, if we notice the debugger state, it would be in break mode as if, while debugging code in Visual Studio, breakpoint has been hit. It is as expected because the captured memory dump represents a point in time state of the process. So, at this point, the **Debug** menu will have the menu items loaded that can be used. For analyzing the dumps of managed

multithreaded applications, the most useful windows are **Parallel Stacks** (**Debug | Windows | Parallel Stacks**):

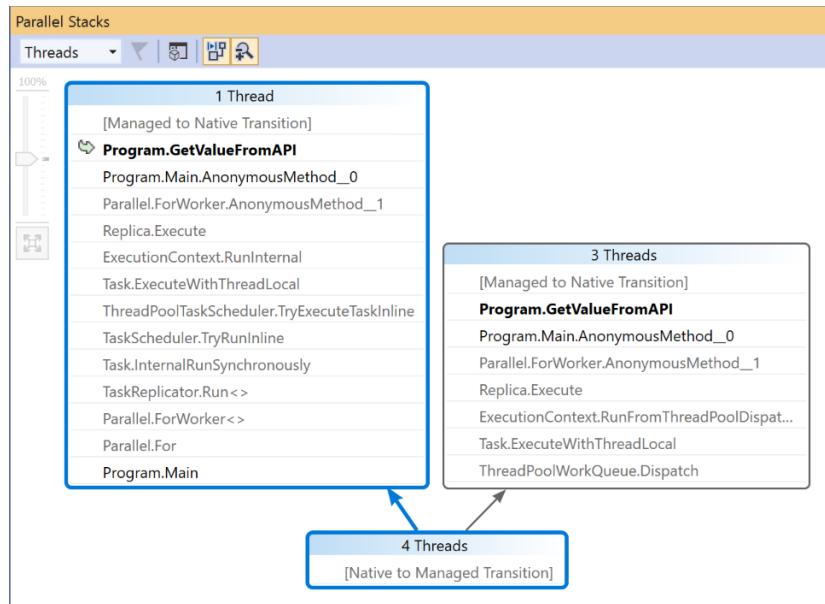


Figure 9.30: Parallel Stacks

It shows that the dump has fourth reads and their call stacks. You can click on at one of these call stacks to see the complete call stack as shown in the following screenshot:

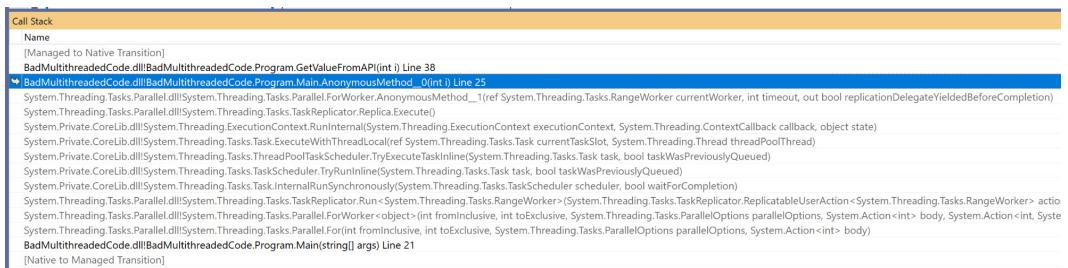


Figure 9.31: Call Stack

We can also see the code map diagram and threads, as shown in the following diagram:

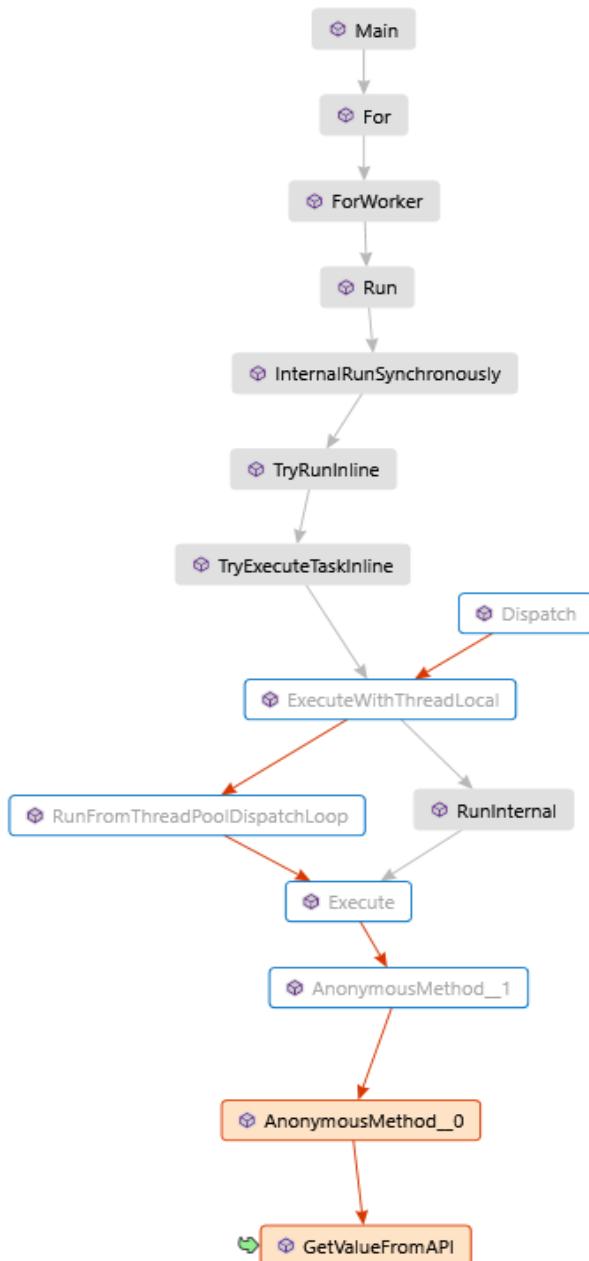


Figure 9.32: Code Map diagram from dump

Threads				Location
	ID	Managed ID	Category	Name
Process ID: cd4848c6-8b57-4fc0-af08-08fa97d43ebe (11 threads)				
▼	4764	0	Main Thread	Main Thread
▼	528	0	Worker Thread	<No Name>
▼	2852	0	Worker Thread	.NET Finalizer
▼	9116	0	Worker Thread	<No Name>
▼	6208	0	Worker Thread	.NET Timer
▼	2956	0	Worker Thread	.NET ThreadPool Worker
▼	4388	0	Worker Thread	.NET ThreadPool Worker
▼	5820	0	Worker Thread	.NET ThreadPool Worker
▼	8668	0	Worker Thread	.NET ThreadPool Worker
▼	4460	0	Worker Thread	.NET ThreadPool Worker
▼	7392	0	Worker Thread	.NET ThreadPool Gate

Figure 9.33: Threads window of the memory dump

Since we have the symbols set correctly, as code was developed in the same machine, apart from the call stack, we also see the code file opened and broke on the line of code when the dump was collected, which helps us identify the issue just like we would while debugging the code!

```

13 ///// Look up data to be populated
14 Dictionary<int, int> lookupData = new Dictionary<int, int>();
15 Console.WriteLine("This sample demonstrates bad code written using multi-threading. We will use
16 // Dumps are collected at a point of time. To make demonstration easier, we will perform operations
17 while (true)
18 {
19     try
20     {
21         Parallel.ForEach(0, 1000000, new ParallelOptions() { MaxDegreeOfParallelism = Environment.
22             {
23                 // Process data returned from API and add it to a collection.
24                 Thread.Sleep(20);
25                 lookupData.Add(i, GetValueFromAPI(i));
26             });
27         }
28     catch (Exception ex)
29     {
30         Console.WriteLine($"Exception occurred in processing data {ex.ToString()}");
31     }
32 }
33
34 static int GetValueFromAPI(int i)
35 {
36     // Simulate delay of 1sec for fetching data from DB/API
37     Thread.Sleep(1000);
38     var result = new Random().Next(0, i);
39     return result;
40 }
41
42 }
```

Figure 9.34: Code view from the memory dump

So, this way, we can identify issues using Visual Studio. Next, we will look at the DebugDiag Analysis tool to analyze the dump.

DebugDiag: One of the most accessible and useful tools to analyze the memory dump is DebugDiag. Its intuitive user interface makes this tool productive for any developer without knowing the intricacies of the memory dump analysis:

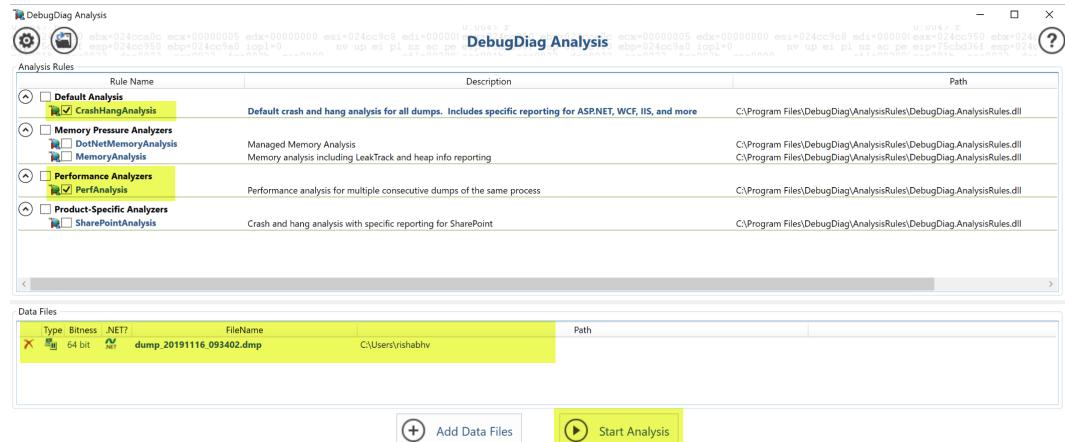


Figure 9.35: DebugDiag

Just launch the DebugDiag Analysis tool, select the analysis rules that we wish to run against the memory dump (highlighted in the section *Analysis Rules*). Then, add the memory dump file to be analyzed by click the **Add Data Files** button. Finally, click on the **Start Analysis** button. It would start the analysis. Till the time analysis is running, a progress screen will be displayed. Once done, it would launch an excellent HTML report, which lists the analysis findings and information. Below is how the report looks like:

Error	
Description	Recommendation
Only mini dumps were selected for analysis. At least one full dump is required for managed (.NET) applications if the analysis is being performed on a machine where the image files are unavailable.	If .NET call stacks are missing or incomplete, include at least one full dump when performing the analysis.

Warning	
Description	Recommendation
The following threads in dump_20191116_093402.dmp are making a call to Sleep API using the .NET Library 0 5 6 9 36.36% of threads blocked (4 threads)	The duration of the Sleep call is unavailable. Please look at the callstack and the code of the function that is calling Sleep to determine the actual time the thread is sleeping.
The following threads in dump_20191116_093402.dmp have evidence of previous .NET exceptions on the stack (0)	Check the Previous .NET Exceptions Report (Exceptions in all .NET Thread Stacks) to view more details of the associated exception

Information	
Description	Recommendation
There are 1 modules which are compiled in DEBUG mode. Please check the List of modules compiled in Debug mode to see those modules and recompile them in Release mode	

Figure 9.36: DebugDiag analysis report

I had run the analysis in a different dump that was collected at the time exception was occurring. Here are the screenshots of the findings from the generated report:

- **Thread Report:**

Top 5 Threads by CPU time

```
Note - Times include both user mode and kernel mode for each thread
Thread ID: 0 Total CPU Time: 00:00:03.765 Entry Point for Thread: BadMultithreadedCode\mainCRTStartup
Thread ID: 5 Total CPU Time: 00:00:00.202 Entry Point for Thread: coreclrThread:intermediateThreadProc
Thread ID: 7 Total CPU Time: 00:00:00.140 Entry Point for Thread: coreclrThread:intermediateThreadProc
Thread ID: 9 Total CPU Time: 00:00:00.124 Entry Point for Thread: coreclrThread:intermediateThreadProc
Thread ID: 8 Total CPU Time: 00:00:00.108 Entry Point for Thread: coreclrThread:intermediateThreadProc
```

.Net Analysis Report

CLR Information

```
CLR version = 4.700.19.46205
Microsoft.Diagnostics.Runtime version = 0.9.2.0
```

.NET Threads Summary

```
Total Threads: 5
Running Threads: 3
Idle Threads: 1
Max Threads: 32767
Min Threads: 4
```

Debugger Thread	Managed Thread ID	OS Thread ID	Thread Object	GC Mode	Domain	Lock Count	Apt	Exception
ID	ID							
0	1	4764	15b34fed0e0	Preemptive	15b35018cc0	0	MTA	
2	2	2852	15b350b3f90	Preemptive	15b35018cc0	0	MTA	(Finalizer)
4	3	6208	15b4ee97b10	Preemptive	15b35018cc0	0	MTA	(Threadpool Worker)
5	4	2956	15b4ef933a0	Preemptive	15b35018cc0	0	MTA	(Threadpool Worker)
6	6	4388	15b4ef9cf10	Preemptive	15b35018cc0	0	MTA	(Threadpool Worker)
7	8	5820	15b4efab8b60	Preemptive	15b35018cc0	0	MTA	(Threadpool Worker)
8	10	8668	15b4ef276f0	Preemptive	15b35018cc0	0	MTA	(Threadpool Worker)
9	5	4460	15b4ef9c8e0	Preemptive	15b35018cc0	0	MTA	(Threadpool Worker)

Figure 9.37: Thread Report

- **Exception Report:**

⊕ Previous .NET Exceptions Report (Exceptions in all .NET Thread Stacks)

Thread ID	Exception Type	Message	Stack Trace
0	System.AggregateException	One or more errors occurred.	System.Threading.Tasks.TaskReplicator.Run[[System.Threading.Tasks.RangeWorker, System.Threading.Tasks.Parallel]]::[ReplicableUserAction`1, System.Threading.Tasks.Parallel]Options, Boolean) + System.Threading.Tasks.ParallelFor[[System.Threading.CancellationToken, System.Private.CoreLib]]::[Int32, Int32, System.Threading.Tasks.Parallel]Options, System.Action`1, System.Action`2, System.Func`4, System.Func`1, System.Threading.Tasks.Parallel.ThrowSingleCancellationExceptionOrOtherException[[System.Collections.ICollection, System.Threading.CancellationToken, System.Exception]] + System.Threading.Tasks.ParallelFor[[Int32, Int32, System.Threading.Tasks.Parallel]Options, System.Action`1, System.Action`2, System.Func`4, System.Func`1, System.Threading.Tasks.ParallelFor[[System._Canon, System.Private.CoreLib]]::[Int32, Int32, System.Threading.Tasks.Parallel]Options, System.Action`1, System.Action`2, System.Func`4, System.Func`1, BadMultithreadedCode.Program.Main[System.String]]

Figure 9.38: Exception Report

Reading this HTML analysis report carefully, helps us find the issues and where they are happening from the dump file.

Windows Debugger (WinDbg): WinDbg is a command-line debugger and probably one of the oldest and powerful debuggers as well. As it accepts commands as inputs, it needs prior knowledge to get started with the dump analysis, but once

you know the fundamentals and the commands, you can dissect the memory dump and find very detailed information. WinDbg can be used to debug both managed and unmanaged memory dumps. As it is one of the oldest, it doesn't know what all modern enhancements have been happening in the world of .NET. To debug managed code, we need a debugger extension called **SOS (Son of Strike)**. This extension contains the commands to analyze the managed memory. So, before starting to analyze the managed memory dump, using WinDbg, we will need to load the SOS extension in the debugger and then run the commands to analyze the dump. There is an extension to SOS called SOSEX, which has additional commands to detect deadlocks, among other useful commands.

If you go to the path `C:\Program Files (x86)\dotnet\shared\Microsoft.NETCore.App\3.1.0` in your Windows machine, you would find a `SOS_README.md` file that contains documentation about the SOS extension and how it is shipped. It leads to the GitHub repository for .NET Core runtime diagnostics tools and their documentation <https://github.com/dotnet/diagnostics#installing-sos>.

Here, we will see a high-level overview of commands to make the reader aware of the most critical and common WinDbg commands and to get started with scenario-specific analysis:

Command	Description
<code>.time</code>	Displays the debug session time, that is, when the dump was collected and the system and process up time at the time dump was taken.
<code>.loadby sos clr</code>	This command is used for analyzing .NET full framework memory dumps. This command tells the debugger to load the <code>sos.dll</code> from the same directory from where <code>CLR.dll</code> was loaded. When running with .NET Core memory dump, you may get an error message, stating unable to find <code>CLR</code> .
<code>.loadby sos coreclr</code>	This command should be used for analyzing .NET Core memory dumps as .NET Core uses CoreCLR. Like the above command, it tells the debugger to load <code>sos.dll</code> from the same directory as CoreCLR.
<code>!sym noisy</code>	Verbose logging of symbols.
<code>lmv</code>	Lists all loaded modules (exes as well as DLLs).
Following commands are useful for investigating hangs, performance issues	
<code>!threadpool</code>	Display CPU usage percentages of the threads.
<code>!Threads</code>	It shows all the managed threads along with their identifiers.

~<ThreadId>s For example, ~2s	Switches the context to the thread whose Thread ID is specified. Thread ID2 in this example.
k	Shows the unmanaged stack trace
!CLRStack	It shows the managed stack trace for the thread in the current context. Use this command to identify what a thread was doing when the dump was collected. Please note, this gives a stack of just one thread that is in context.
~*e !CLRStack	This command shows the call stack of all the threads.
!runaway	Lists all the threads with the time they have been running. Use this command to identify if there is any hang or deadlock, as in such cases, the thread would run for a long time.
!syncblk	Tells us the threads that are waiting on monitors or locks. Use this command to identify threads that are blocked, so helpful in investigating performance and deadlock scenarios.
!dumpheap - thinlock	Displays all the locks that do not have conflicts.
Following commands are useful for investigating crash issues	
!analyze -v	Displays the exception information with the verbose switch (-v), so we get very detailed information. Useful to identify if any exceptions are there in the memory dump.
Following commands are useful for investigating high memory issues	
!eeheap -gc	Displays information on memory heaps used by the garbage collector.
!dumpheap -stat	Displays the statistics of objects and memory that they are consuming. The first column output is referred to as the method table, which is an index to the type of object.
!dumpheap -mt methodtable	This command displays all objects of the type (based on method table, returned from the above command, or otherwise). The first column is referred to as address.
!do address	This command is dump object, and as the name suggests, it dumps the object and displays properties of that object, including value.
!gcroot address	This command identifies what objects reference the specified address. Useful to identify why objects are not collected.
du value	This command converts the value obtained from (!do address) into a readable format.

The screenshots of a few of the preceding commands and their output are displayed in the next screenshot:

```

Command x
0:010> .time
Debug session time: Sat Nov 16 09:34:03.000 2019 (UTC + 0:00)
System Uptime: not available
Process Uptime: 0 days 1:45:46.000
Kernel time: 0 days 0:00:01.000
User time: 0 days 0:00:02.000
0:010> !loadby sos coreclr
0:010> !threaddpool
CPU utilization: 25Unknown format characterUnknown format control characterWorker Thread: Total: 5 Running: 3 Idle: 1 MaxLimit: 32767 MinLimit: 4
Work Request in Queue: 0
-----
Number of Timers: 0
-----
Completion Port Thread:Total: 0 Free: 0 MaxFree: 8 CurrentLimit: 0 MaxLimit: 1000 MinLimit: 4
0:010> !threads
ThreadCount: 8
UnstartedThread: 0
BackgroundThread: 7
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
-----
Lock
DBG ID OSID ThreadObj State GC Mode GC Alloc Context Domain Count Apt Exception
0 1 129c 0000015B338181AE3 0000015B38183610 0000015b35018cc8 0 MTA
2 2 b24 0000015B350B3F90 202a20 Preemptive 0000000000000000:0000000000000000 0000015b35018cc8 0 MTA (Finalizer)
4 3 1840 0000015B4EE97810 102a20 Preemptive 0000000000000000:0000000000000000 0000015b35018cc8 0 MTA (Threadpool Worker)
5 4 b8c 0000015B4EF93340 302a20 Preemptive 0000015B380DB810:0000015B380DBF88 0000015b35018cc8 0 MTA (Threadpool Worker)
6 6 1124 0000015B4EF9C18 302a20 Preemptive 0000015B380FFA0:0000015B380FFD48 0000015b35018cc8 0 MTA (Threadpool Worker)
7 8 16bc 0000015B4EF8A860 102a20 Preemptive 0000015B38121A8:0000015B38121BC8 0000015b35018cc8 0 MTA (Threadpool Worker)
8 10 21dc 0000015B4EF276F0 102a20 Preemptive 0000015B37FA8078:0000015B37FA90C8 0000015b35018cc8 0 MTA (Threadpool Worker)
9 5 110c 0000015B4EF9C8E0 302a20 Preemptive 0000015B37F872A0:0000015B37F872A8 0000015b35018cc8 0 MTA (Threadpool Worker)
0:010> !runaway
User Mode Time
Thread Time
0:129c 0 days 0:00:02.312
5:b8c 0 days 0:00:00.171
9:116c 0 days 0:00:00.078
8:21dc 0 days 0:00:00.062
7:16bc 0 days 0:00:00.062
6:1124 0 days 0:00:00.031
2:b24 0 days 0:00:00.015
10:ab4 0 days 0:00:00.000
4:1840 0 days 0:00:00.000
3:239c 0 days 0:00:00.000
1:210 0 days 0:00:00.000
0:010> +e !CLRStack
05 Thread Id: 0x129c (0)
Child SP IP Call Site
00000004D517DBF8 00007ffff9eb1ffde4 [HelperMethodFrame: 00000004d517dbf8] System.Threading.Thread.SleepInternal(Int32)
00000004D517DC0F 00007ffff946abbhd2 BadMultiThreadedCode.Program.GetValueFromAPI(Int32) [C:\Users\vishabhv\source\repos\Parallel-Programming-with-C-8-and-9\Parallel-Programming-with-C-8-and-9\Program.cs:line 10]
00000004D517DD40 00007ffff946abb66 BadMultiThreadedCode.Program+<DisplayClass_0_b_0>(Int32) [C:\Users\vishabhv\source\repos\Parallel-Programming-with-C-8-and-9\Parallel-Programming-with-C-8-and-9\Program.cs:line 11]
00000004D517DE00 00007ffff946adab7c System.Threading.Tasks.Parallel+<DisplayClass19_0>(Int32) [C:\Users\vishabhv\source\repos\Parallel-Programming-with-C-8-and-9\Parallel-Programming-with-C-8-and-9\Program.cs:line 12]
00000004D517DE20 00007ffff946adab74 System.Threading.Tasks.TaskReplicator+<Execute>C<T> [/src/System.Threading.Tasks.Parallel/src/System.Threading.Tasks.TaskReplicator.cs:line 13]
00000004D517DE40 00007ffff946cb201 System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext, System.Threading.ContextCallback, System.Threading.CancellationToken, System.Threading.Tasks.Task<T>) [/src/System.Threading.ExecutionContext.cs:line 14]
00000004D517DF10 00007ffff946adab14 System.Threading.Tasks.Task+<ExecuteWithTheLocal>C<T> [/src/System.Threading.Tasks.Task.cs:line 15]
00000004D517DFC0 00007ffff946cb1ef8 System.Threading.Tasks.TasksPoolTaskScheduler.TryExecuteTaskInline(System.Threading.Tasks.Task<T>, Boolean) [/src/System.Threading.Tasks.TasksPoolTaskScheduler.cs:line 16]
00000004D517E010 00007ffff946cb116 System.Threading.Tasks.TaskScheduler+<TryRunInline>C<T> [/src/System.Private.CoreLib/src/System.Threading.Tasks.TaskScheduler.cs:line 17]
00000004D517E050 00007ffff946cb1d9 System.Threading.Tasks.Task.InternalRunSynchronously(System.Threading.Tasks.TaskScheduler, Boolean) [/src/System.Private.CoreLib/src/System.Threading.Tasks.Task.cs:line 18]
00000004D517E0A0 00007ffff946cb1b20 System.Threading.Tasks.TaskReplicator+<Run>C<T> [/src/System.Threading.Tasks.TasksPoolTaskScheduler.cs:line 19]
00000004D517E100 00007ffff946cb1680 System.Threading.Tasks.TaskParallelForWorker+<Run>C<T> [/src/System.Private.CoreLib/src/System.Threading.Tasks.ParallelForWorker.cs:line 20]
00000004D517E1C0 00007ffff946cb1287 System.Threading.Tasks.ParallelFor<T>(Int32, Int32, System.Threading.Tasks.ParallelOptions, System.Action`1) [/src/System.Threading.Tasks.ParallelFor.cs:line 21]
00000004D517E230 00007ffff946cb1a94 BadMultiThreadedCode.Program.Main(System.String[]) [C:\Users\vishabhv\source\repos\Parallel-Programming-with-C-8-and-9\Parallel-Programming-with-C-8-and-9\Program.cs:line 22]
00000004D517E4E8 00007ffff9a65a3903 [GCFrame: 00000004d517e4e8]
00000004D517EA90 00007ffff9a65a3903 [GCFrame: 00000004d517ea90]
05 Thread Id: 0x210 (1)

```

Figure 9.39: WinDbg commands

The essential stuff commands and stack trace is highlighted. We see that !threads command displays the details of all the threads. For each thread, we see its identifier, its state, apartment, as well as lock count. This lock count helps us identify if threads are blocked due to a monitor or lock statement and also possible deadlocks. In the above screenshot, there are no threads that are locked.

For a detailed set of commands, I would recommend reading the MSDN blogs from Tess Fernandes (who is one of the great exponents of WinDbg and debugging in general) at <https://blogs.msdn.microsoft.com/tess/>. Also, the official documentation of WinDbg is a must-read for all the enthusiastic developers wanting to learn to

debug at <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/>. Also, the scope of this discussion was limited to Windows. If you wish to get started with managed debugging in Linux, this excellent post from the MSDN blog is a great start <https://devblogs.microsoft.com/premier-developer/debugging-net-core-with-sos-everywhere/>.

dotnet-dump: Apart from collecting the memory dump, this command-line tool from .NET Core SDK can be used to analyze the dump as well. This tool also supports the SOS commands that we just discussed in the above section. The best way to get started with analysis using **dotnet-dump** is to use the **help** command, which lists the commands and their usage, as shown in the following diagram:

```
C:\Users\rishabhv>dotnet-dump analyze "C:\Users\rishabhv\dump_20191116_093402.dmp"
Loading core dump: C:\Users\rishabhv\dump_20191116_093402.dmp ...
Ready to process analysis commands. Type 'help' to list available commands or 'help [command]' to get detailed help on a command.
Type 'quit' or 'exit' to exit the session.
> help
Usage:
  > [command]

Commands:
  cldmodules          Lists the managed modules in the process.
  exit, quit          Exit interactive mode.
  help, soshelp <command> Display help for a command.
  logging             Enable/disable internal logging
  lm, modules         Displays the native modules in the process.
  registers <threadindex> Displays the thread's registers.
  threads, setthread <threadindex> Sets or displays the current thread for the SOS commands.
  dumprcw <arguments> Displays information about a Runtime Callable Wrapper.
  dumpccw <arguments> Displays information about a COM Callable Wrapper.
  dumppermissionset <arguments> Displays a PermissionSet object (debug build only).
  traverseheap <arguments> Writes out a file in a format understood by the CLR Profiler.
  analyzeoom <arguments> Displays the info of the last OOM occurred on an allocation request to the GC heap.
  verifyobj <arguments> Checks the object for signs of corruption.
  listnearobj <arguments> Displays the object preceding and succeeding the address specified.
  gcheapstat <arguments> Display various GC heap statistics.
  watsonbuckets <arguments> Displays the Watson buckets.
  threadpool <arguments> Lists basic information about the thread pool.
  comstate <arguments> Lists the COM apartment model for each thread.
  gchandle <arguments> Provides statistics about GCHandles in the process.
  objsize <arguments> Lists the sizes of the all the objects found on managed threads.
  gchandleleaks <arguments> Helps in tracking down GCHandle leaks.
  clrstack <arguments> Provides a stack trace of managed code only.
  clrthreads <arguments> List the managed threads running.
  dumparray <arguments> Displays details about a managed array.
  dumpasync <arguments> Displays info about sync state machines on the garbage-collected heap.
  dumpassembly <arguments> Displays details about an assembly.
  dumpclass <arguments> Displays information about a EE class structure at the specified address.
  dumpdelegate <arguments> Displays information about a delegate.
  dumpdomain <arguments> Displays information all the AppDomains and all assemblies within the domains.
  dumpheap <arguments> Displays info about the garbage-collected heap and collection statistics about objects.
  dumpil <arguments> Displays the Microsoft intermediate language (MSIL) that is associated with a managed method.
  dumplog <arguments> Writes the contents of an in-memory stress log to the specified file.
  dumpmd <arguments> Displays information about a MethodDesc structure at the specified address.
  dumpmodule <arguments> Displays information about a EE module structure at the specified address.
  dumpmt <arguments> Displays information about a method table at the specified address.
  dumpobj <arguments> Displays info about an object at the specified address.
  dumpvc <arguments> Displays info about the fields of a value class.
  dso, dumpstackobjects <arguments> Displays all managed objects found within the bounds of the current stack.
  eeheap <arguments> Displays info about process memory consumed by internal runtime data structures.
  eeversion <arguments> Displays information about the runtime version.
  finalizequeue <arguments> Displays all objects registered for finalization.
  gcref <arguments> Displays info about references (or roots) to an object at the specified address.
  gcwhere <arguments> Displays the location in the GC heap of the argument passed in.
  ip2md <arguments> Displays the MethodDesc structure at the specified address in code that has been JIT-compiled.
  name2ee <arguments> Displays the Methodable structure and EECClass structure for the specified type or method in the assembly.
  pe, printexception <arguments> Displays and formats fields of any object derived from the Exception class at the specified address.
  sosstatus <arguments> Displays the global SOS status.
  syncblk <arguments> Displays the SyncBlock holder info.
  histclear <arguments> Releases any resources used by the family of Hist commands.
  histinit <arguments> Initializes the SOS structures from the stress log saved in the debuggee.
  histobj <arguments> Examines all stress log relocation records and displays the chain of garbage collection relocation arguments.
  histobjfind <arguments> Displays all the log entries that reference an object at the specified address.
  histroot <arguments> Displays information related to both promotions and relocations of the specified root.
  setsymbolserver <arguments> Enables the symbol server support.
```

Figure 9.40: **dotnet-dump analyze help** command

Then, we can run the commands and analyze the memory dump file to investigate the issue. This section is just an introduction to this tool. For a detailed discussion on

commands to analyze the dump using this tool, please read the documentation of analyzing command at <https://docs.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-dump#dotnet-dump-analyze>.

In the last two sections, we discussed the memory dump collection and analysis process. It is a convenient way to investigate and troubleshoot issues in production. Using memory dumps, we can investigate crashes, hangs, deadlocks, performance issues, as well as high CPU issues. Let's conclude our discussion by fixing the wrong code.

Fixing

So far, we discussed the memory dump collection and analysis process. Here it is important to note that memory dumps represent the state of the process at a given time, and if at that instance, the issue does not occur, then dump analysis would not yield fruitful results. So, it's essential to understand this fact and explore the possibilities to collect multiple dumps so that analysis can lead to a conclusion. I collected a couple of memory dumps for this simple demonstration of the issue, and in one of the dumps, I found the issue, while the other one didn't have any issue. Once the issue is found, fixing generally doesn't take much time. But I highly encourage all the readers to give priority to quality, and hence they should thoroughly perform unit testing of any code they write and apply all the scenarios, including edge cases, to fix issues properly in code. It's always better to find your own mistakes and correct them, rather than having someone else point it.

Based on memory dump analysis, as well as code review, it is easy to point out that the sample code makes incorrect use of updating a `Dictionary<int, int>` inside a multithreaded `Parallel.For` loop. Recall that Dictionary, list, and many more, collections are not thread-safe. The exceptions from the memory dump also point to this and even list the line number where the issue is suspected. This makes the fix, really simple, just use `ConcurrentDictionary<int, int>` instead of `Dictionary<int, int>` and issue is fixed.

The working version of the code is as shown in the following code:

```
static void Main(string[] args)
{
    //// Look up data to be populated
    ConcurrentDictionary<int, int> lookupData = new
    ConcurrentDictionary<int, int>();
    Console.WriteLine("This sample demonstrates bad code written
using multi-threading. We will use this sample learn to collect and
analyze dumps via different tools !");
    // Dumps are collected at a point of time. To make
```

demonstration easier, we will perform operations inside an infinite loop.

```
    while (true)
    {
        try
        {
            Parallel.For(0, 1000000, new ParallelOptions() {
MaxDegreeOfParallelism = Environment.ProcessorCount }, (i) =>
{
                ///// Process data returned from API and
added it to a collection.

                Thread.Sleep(20);
                lookupData.TryAdd(i,
GetValueFromAPI(i));
            });
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception occurred in processing
data {ex.ToString()}");
        }
    }
}
```

Performance is an essential aspect of modern software and one of the compelling reasons to use multithreading. In the next section, we will learn to investigate performance issues with a great tool called PerfView.

Performance analysis with PerfView

Earlier in this chapter, we used a sample WPF app and profiled it using Visual Studio to identify performance bottlenecks before shipping our product. However, it may well happen that while in production, you observe a performance issue in your application, which was either missed during development or not identified in the testing. Or consider a scenario in which an application is developed by some other company and shipped to the customer. The customer observes performance glitches and asks you to investigate the cause of performance issues in that application. How would you approach such a case if you do happen to take up this job? Yes, you can capture memory dumps, analyze them, and share finding with your customers. However, there is a great tool to identify performance issues in .NET called PerfView.

This tool is developed by an architect named Vance Morrison in Microsoft. It works equally well with .NET Core as well with whatever little I have used it with .NET Core. We will wrap up this chapter by identifying the performance issue of our WPF app, but this time via PerfView. We will follow the below steps:

1. Launch PerfView. For the first time use, you would need to agree to a license agreement.
2. In the top menu bar, click on **Collect | Run**. (*Alt + R*):

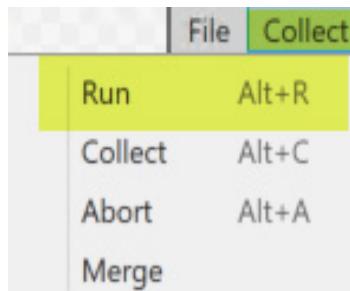


Figure 9.41: Run (*Alt + R*)

3. It will open a dialog with few form fields. Enter the path to the executable that we wish to run in the Command. PerfView performs the machine-wide collection, so we can also run PerfView later after starting the process as you may please. Steps here are just to enable the reader to get started with PerfView:

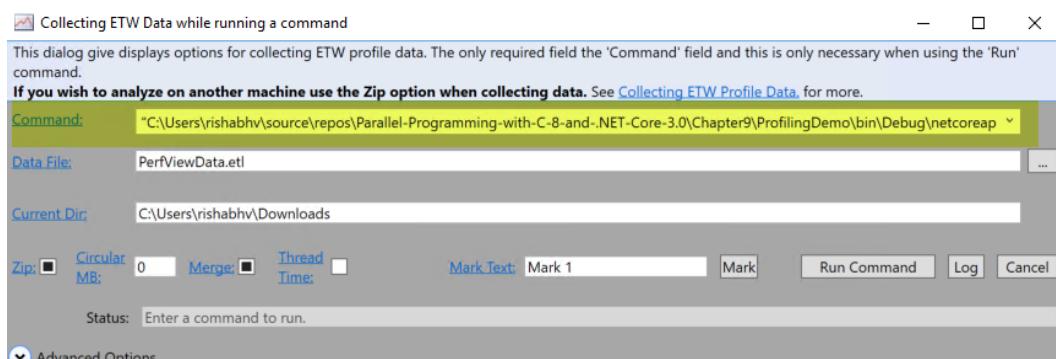


Figure 9.42: Run Dialog

4. It will launch the WPF app. Click on the **Download** button to reproduce the issue. Once the issue is reproduced, you can close the application. PerfView has already started collecting the traces and may prompt you with a couple of dialogs for symbol server paths, which you should answer in the affirmative. After some time, the collection would complete, and traces would be merged

to etl.zip file, which PerfView will display in its left panel as shown in the following screenshot:

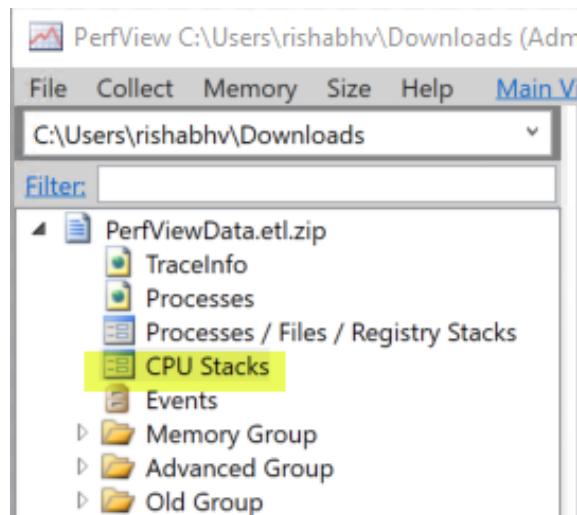


Figure 9.43: ETL.zip file

5. Though, the reader should click on each of the items displayed in the previous image and see for themselves what each of them contains. For this demonstration, we would click on CPU Stacks. As mentioned earlier, PerfView collects the machine-wide traces, that is, traces for all the processes, so the dialog will list all the running processes, of which we will choose the WPF app that we are investigating:

The screenshot shows the 'Select Process Window' dialog box. The text area at the top states: 'The data that was collect was machine wide, but typically you are interested in only one process. This dialog box allows you to choose a particular process to focus on by either double clicking on a process or by selecting a process and hitting OK. The 'All Procs' button can be used if you wish to look at all data. See [Selecting A Process Help](#) for more.' Below this is a table listing processes:

Name	ID	Parent	CPU MSec	Start	Duration	CommandLine
ProfilingDemo	11644	9840	3,546	19-11-17 21:23:10	19.68 sec	"C:\Users\rishabhv\source\repos\WPFApp1\bin\Debug\WPFApp1.exe"
PerfView	9840	7212	799	19-11-17 21:23:09	23.96 sec	"C:\Users\rishabhv\Downloads\PerfViewData.etl.zip"
dwm	6024	3032	614	19-11-17 21:23:09	23.96 sec	"dwm.exe"
svchost	2084	676	591	19-11-17 21:23:09	23.96 sec	C:\Windows\System32\svchost.exe
devenv	4656	4824	470	19-11-17 21:23:09	23.96 sec	"C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\Common7\IDE\devenv.exe"
MsMpEng	3420	676	298	19-11-17 21:23:09	23.96 sec	"C:\ProgramData\Microsoft\Windows\Shared\LowPriorityBackgroundTasks\msmpeng.exe"
System	4	-1	265	19-11-17 21:23:09	23.96 sec	
svchost	1096	676	143	19-11-17 21:23:09	23.96 sec	C:\Windows\System32\svchost.exe
ServiceHub.RoslynCodeAnalysisService32	8480	8508	140	19-11-17 21:23:09	23.96 sec	"C:\Program Files (x86)\Microsoft\Visual Studio\2017\Community\MSBuild\bin\Roslyn\servicehub.exe"
PerfWatson2	700	4656	132	19-11-17 21:23:09	23.96 sec	"C:\Program Files (x86)\Microsoft\Visual Studio\2017\Community\MSBuild\bin\Roslyn\perfwatson.exe"
explorer	4824	3892	101	19-11-17 21:23:09	23.96 sec	C:\Windows\Explorer.EXE
Idle	0	-1	59	19-11-17 21:23:09	23.96 sec	

At the bottom of the dialog are three buttons: 'OK', 'All Procs', and 'Cancel'.

Figure 9.44: Select process dialog

6. I clicked on ProfilingDemo (the application to be profiled) and then the OK button. It opens the CPU stacks for the selected process and lists the functions doing most work. I say doing most work because PerfView collects sampled traces of methods, so if a method is running for a small duration in the range, it has a lesser number of samples. If the method is running for a longer time, more samples would be collected, so, generally, the function that is invoked most or runs the longest would have a higher number of samples. It is a very naive simplification to make the reader understand the working of PerfView. Though our WPF app code is not drastically wrong, we can still figure out from this dialog that the issue is in the BadCode method called from BtnDownload_Click method. And this is the same and right finding that we had from profiling the application as well:

Name	Exc %	Exc	Inc %	Inc	Fold	When	First	Last
profilingdemo!	0.8	25	66.5	2,029.0	25	02_09710003_49992_ooo00o1	589.003	9,305.877
profilingdemo!ProfilingDemo.MainWindow.BtnDownload_Click()	0.0	1	39.1	1,194.0	1	0_3_49991	2,101.156	5,862.527
profilingdemo!ProfilingDemo.MainWindow.BadCode()	0.0	1	39.1	1,193.0	1	0_3_49991	2,101.156	5,862.527
Thread (8228) CPU=71ms (.NET Finalizer)	0.0	0	2.3	71.0	0	0_01	686.013	9,281.874
Thread (9360) CPU=139ms (.NET IO Thread Pool Worker)	0.0	0	4.6	141.0	0	0_3_100	2,097.152	4,754.421
Thread (10104) CPU=197ms (.NET Thread Pool Worker)	0.0	0	6.4	196.0	0	0_01_0_20_01	2,723.239	7,761.721
Thread (468) CPU=354ms (.NET Thread Pool Worker)	0.0	0	11.5	351.0	0	0_1100_1_1_3	2,121.173	5,739.524
ROOT	0.0	0	100.0	3,051.0	0	03_0A912417_124CBG4_0100002	583.002	9,330.875
profilingdemo!ProfilingDemo.MainWindow.InitializeComponents()	0.0	0	5.1	155.0	0	31	1,730.126	1,891.135
profilingdemo!ProfilingDemo.MainWindow.ctor()	0.0	0	5.5	167.0	0	32	1,727.116	1,900.135
Thread (10624) CPU=60ms	0.0	0	2.0	60.0	0	0_0_oo_0_oo_0	1,774.138	9,271.880
profilingdemo!ProfilingDemo.App.Main()	0.0	0	61.5	1,875.0	0	09710003_49992_ooo00o0	1,517.113	9,280.886
Thread (9888) CPU=2107ms (Startup Thread)	0.0	0	68.8	2,100.0	0	02_09710003_49992_ooo00o2	583.002	9,330.875
profilingdemo!ProfilingDemo.App.ctor()	0.0	0	4.4	135.0	0	04	1,517.113	1,652.111
Thread (988) CPU=37ms (.NET Thread Pool Worker)	0.0	0	1.4	43.0	0	0_0_oo_0_oo	5,141.458	8,870.832

Figure 9.45: CPU stack

Though a reader may think that while we did the performance analysis, only the **Download** button was clicked, so isn't it expected. The answer is yes, it is expected. But the intention here is to demonstrate how PerfView can be used to trace down performance issues. What we have seen is an over-simplistic example to understand the tool. The tool is very exhaustive and has lots of features that are very well documented in the tool itself. Anytime you feel like you are lost while using the tool, you can click on the hyperlinks on the tool, they will all redirect you to the relevant help page, which will answer your queries. Apart from this, tips and troubleshooting procedure is also a hyperlink and well documented in help, so I would highly encourage the readers to spend some time reading it.

To know and understand about PerfView in details, the following links would be useful as well apart from help and documentation:

- <https://github.com/Microsoft/perfview>
- Video tutorials at <http://channel9.msdn.com/Series/PerfView-Tutorial>

- Vance Morrison's blog <http://blogs.msdn.com/b/vancem/archive/tags/perfview>

Summary

In this chapter, we discussed the debugging and troubleshooting fundamentals, tools, and techniques using Visual Studio 2019 and related tools. We profiled an application using Visual Studio profiler. We learned about memory dumps, their types, and what information they contain, and because of the information they contain, they can be used for post-mortem debugging. We then discussed the tools and techniques to collect and analyze the memory dumps, and using that analysis fixed our bad code sample. Finally, we learned about troubleshooting performance issues using the PerfView tool. In the next chapter, we will see some handy and cooling tips and tricks for multithreaded and parallel programming.

Exercise

1. Explore, research, and learn time travel debugging using:
 - a. IntelliTrace in Visual Studio
 - b. WinDbg
2. How can you switch from the command window to the immediate window and vice versa in Visual Studio?
3. Write a .NET Core 3.1 console app that goes into an infinite loop. HINT: Recursion with no terminating condition.
4. Run the program developed in the previous problems, and using different tools that we discussed, capture the memory dumps.
5. Now analyze the dumps collected in the previous problem, using:
 - a. DebugDiag
 - b. Visual Studio 2019
 - c. WinDbg
6. Are you able to identify the issue just by analyzing the dumps? Fix the code.
7. Read the links and official documentation of WinDbg and its commands (Links shared in the discussion)
8. Watch the video tutorial of PerfView at <http://channel9.msdn.com/Series/PerfView-Tutorial>.

CHAPTER 10

Tips and Tricks

"If I have seen further than others, it is by standing upon the shoulders of giants."

- Sir Issac Newton

Parallel and multithreaded programming is a delicate art to master. Even with deliberate coding, there may be issues and unwarranted results. Good thing, though, is that this programming paradigm has been in use for a while and so we can avoid the common pitfalls by following the learnings, tips, tricks, and best practices from the previous implementations. In this chapter, we will discuss the tips, tricks, and best practices that will help us write better-multithreaded applications. The development of .NET Core applications is done using Visual Studio IDE and/or Visual Studio Code editor so that we will see a few tips and tricks on Visual Studio as well. .NET / .NET Core and Azure is a marriage made in heaven, and so it is no surprise that most of the .NET / .NET Core-based web applications are deployed in Azure, so we would also discuss the tips and tricks for developing, deploying and monitoring multithreaded applications in Azure.

Structure

We will cover the following topics in this chapter:

- Tips and tricks
- Summary

Objectives

By the end of this chapter, the reader should be able to:

- See and benefit from the tips and tricks in writing better-multithreaded applications
- Leverage tips and tricks to be more productive in coding and development

Tips and tricks

Most developers, I come across, struggle to understand .NET, .NET Core, .NET Native, .NET Standard, so a quick recap of them:

- **.NET**: Framework developed by Microsoft, which runs primarily on Windows. It has CLR that manages the execution, management of code, a **framework class library (FCL)** that provides a rich set of APIs to build the applications.
- **.NET Core**: A truly cross-platform framework developed by Microsoft (rewritten implementation of .NET to work across platforms). It has a cross-platform implementation of **CLR** called **CoreCLR** and streamlined implementation of class libraries called **CoreFx**.
- **.NET Native**: The default implementation of .NET is that the compiler compiles the C# code to **Microsoft Intermediate Language (MSIL)** or IL, and then at the time of execution, this IL is **just in time (JIT)** compiled to native assembly level instructions. It has its advantages, but it generally makes things a little slower and takes up memory. There is another initiative called .NET Native, in which the C# code is directly compiled to native CPU instructions **ahead of time (AoT)**. It makes code execute faster and consumes less memory. It finds use in HoloLens, Xbox One, and many more.
- **Mono**: Mono is a third-party cross-platform implementation of .NET, but it fell behind in the implementation of the Microsoft implementation of .NET and didn't take off in a significant way. It was with Xamarin (platform for Mobile development) that Mono found some traction.
- **.NET Standard**: Think of it as a standard contract which lays down a set of APIs that all the .NET platforms must implement. When we say .NET Core 3.1 implements .NET Standard 2.1, it means .NET Core 3.1 will have

all the APIs that are defined in .NET Standard 2.1 and likewise, if any implementation of .NET says that it implements .NET Standard 2.1, then that .NET implementation is guaranteed to have all these APIs as well.

As of writing this chapter, .NET Core 3.1 is going to be the last version of .NET Core. Post that, there would be .NET 5 releasing in late 2020. It will remove the need of having to use .NET Core for making the cross-platform applications. The roadmap of .NET Core can be seen at <https://github.com/dotnet/core/blob/master/roadmap.md>.

Threading and TPL

- Thread creation is expensive. Avoid creating them unless you absolutely must for executing a long-running task.
- More threads don't mean better performance. Always perform benchmarking with and without multithreading, to understand if multithreading is helping in the scenario or not.
- All threads created by default are foreground threads, unless we programmatically make them as background, by setting `IsBackground=true`.
- An application must have at least one foreground thread running for the application to continue running. If all the foreground threads terminate, then background threads, if any, would stop and not run to completion.
- Prefer using `ThreadPool` than explicitly creating and managing the threads.
- TPL should be preferred for performing CPU intensive tasks.
- Prefer using `CancellationToken` to cancel the operation rather than custom implementation.
- Do not use Parallel loops for a smaller number of iterations. The cost of context switching may outweigh the benefits of parallelism.
- Avoid writing to a shared resource inside the Parallel loop.
- Avoid using locking inside Parallel loops as it negates the benefit of parallelism.
- Prefer using thread-safe collections that ships with TPL in multithreaded programming, instead of regular collections.

async await

The `async void` is for top-level event handlers only. Avoid using `async void` while defining your `async` methods. Always prefer `async Task` instead of the `async void`, as shown below:

```
private async void Button_Click(object sender, EventArgs e)
```

```
{  
    // DO NOT USE - NOT RECOMMENDED  
}  
  
private async Task Button_Click(object sender, EventArgs e)  
{  
    // PREFER async Task not async void  
}
```

- While using `async await`, ensure it is used from top to bottom in the entire call hierarchy, that is, all the method calls from top to bottom should be `async`. Another way of saying the same thing is, use `async` all the way.
- Avoid blocking `async` methods at all costs. Do not use the following:
 - `Task.Wait`
 - `Task.Result`
 - `Task.WaitAny`
 - `Task.WaitAll`
 - `Thread.Sleep`
- Avoid `sync` over `async`, as it is essentially blocking the `async` method.
- Avoid `async` over `sync` as it is faking or lying to the API consumer, and it would lead to scalability issues. On a lighter note, the name of the class library is not lie-brary! So, don't lie.
- Always identify if the work you want to perform is compute-bound or I/O bound before deciding to use `async await` or `ThreadPool`.
- Compute bound work is one that uses CPU extensively, like an extensive iteration, or big computations inside the loop, or LINQ over objects.
- I/O bound work, as the name suggests, is the operation that does file or network I/O.
- For CPU bound work, consider using background threads or `Task.Run` or Parallel constructs to schedule the operation to the `ThreadPool`. `Task.Run` should preferably be used only if the method is not exposed externally.
- For I/O bound work, prefer using `async await` instead of background threads.
- Always consider using the `task.ConfigureAwait(false)` if you are exposing the API or method from a class library.
- Always suffix `Async` in the `async` method names, to make them easily identifiable.

- Avoid wrapping the code inside using block if the body makes a call to the `async` methods. It is because, if the invocation is not awaited, the wrapping object may dispose before the call finishes. Hence always `await async` methods!
- When working on graphical user interfaces (WPF apps), ensure that your code in such a way that your UI thread remains free as much as possible. Don't do any long-running operations on the UI thread. Doing so will kill user experience and make your UI unresponsive.
- `async await` supports the usage of `await` in `catch` block as well, so if you have `async` APIs to log the exception, do make use of them.
- Always consider performing benchmarking of your application using tools like Benchmark.NET <https://github.com/dotnet/BenchmarkDotNet>.

ASP.NET Core

- `ThreadPool` starvation is a situation where there are not enough threads available to process the requests. One of the primary reasons for starvation is that threads are unable to gain access to a shared resource and hence unable to make progress and hence stays blocked and cannot return to the pool. It generally can happen, if the shared resource is made unavailable for an extended period, due to greedy long-running code. Consider, for example, a piece of code that is wrapped around a lock statement. If this piece of code takes a long time to complete, then all the other threads trying to execute this piece of code would remain blocked for an extended period. In such cases, if a burst of request comes to the server, then the `ThreadPool` would throttle the creation of threads, and there may be HTTP 503 errors.
- ASP.NET Core, unlike ASP.NET, does not have a request queue. In ASP.NET, there is a `RequestQueue` class that resides in `System.Web` namespace. This class is responsible for preventing `ThreadPool` starvation in ASP.NET. This class doesn't exist in ASP.NET Core, and so it is susceptible to `ThreadPool` starvation if code is not written correctly. So, avoid blocking calls.
- Do log `ThreadPool` statistics if your application handles many requests.
- Consider caching the static data for better performance.
- ASP.NET Core doesn't have `SynchronizationContext`. So, `task.ConfigureAwait(false)` doesn't make any difference in ASP.NET Core. The good thing because of this change is that even with lousy code like the `task.Wait()` or `task.Result`, there cannot be deadlock. But this should not be taken as the license to use blocking calls.
- ASP.NET Core doesn't have `AppDomain`, unlike ASP.NET. Just think that in ASP.NET Core, there is a single process and single `AppDomain`. For

the dynamic loading of assemblies, the recommendation is to use the `AssemblyLoadContext` class.

- `HttpContext` is not thread-safe. So, accessing or modifying the `HttpContext` concurrently or in parallel may lead to unreliable data.
- Task continuations in ASP.NET Core are queued to the `ThreadPool` and so can run in parallel (not in continuation like one after another) and hence may not work as expected. So, do ensure you test your code multiple times to avoid any unwarranted bugs or glitches.
- Consider using tools like `BlockingDetector` to detect blocking calls. The tool can be seen at <https://github.com/benaadams/Ben.BlockingDetector>.

Threading Patterns

- If you are building a new application, always use a **Task-based Asynchronous Pattern** for implementing asynchronous methods as `Task` gives a way to handle cancellation, continuation, report progress, secure exception handling.
- Never use `GetAwaiter().GetResult()` to retrieve data from an `async` method in the asynchronous method. It should always and only be awaited hence should `async` all the way.
- Always create `async` methods with cancellation token as one of the parameters. It helps in gracefully canceling the execution of the method.
- Coordination between tasks can be achieved through constructs, such as the `Task.WhenAll`, `Task.WhenAny()` (avoid `Task.WaitAll` `Task.WaitAny`) and should be used along with `await`.

Synchronization

- Do not use `lock(this)` or `lock(typeof(T))` as it locks the entire instance and type, respectively.
- Use locks with caution as locking on interdependent object instances may lead to deadlocks.
- When in doubt, consider using `Monitor.TryEnter` inside try block and `Monitor.Exit` inside finally block, to provide a timeout value. This timeout can prevent the infinite lock of resources and hence can alleviate the deadlock.
- When working with value types like `int` in multithreaded code, consider making use of atomic operations leveraging the `Interlocked` APIs.
- Synchronization among threads should be done using one of the synchronization constructs like `Mutex`, `Semaphore`/`SemaphoreSlim` to name a few.

- Mutex and semaphore are massive constructs and can be used for inter-processor machine-wide synchronization as well.
- Critical sections in code should always use synchronization constructs to prevent data corruption.
- C# provides numerous concurrent collections that can be used in the case of shared objects.
- Try to use Command Query Segregation Principle, which means separate models for read and write.
- Lock statements would not work with async await statements, so consider using SemaphoreSlim, rather than sacrificing asynchrony or thread-safety.

Testing

- Testing philosophy should always be *In God we trust, rest all we test*. So, don't make any assumptions and always test all the features without any bias and irrespective of which developer has developed it.
- Don't ever ship your application without conducting proper unit testing, integration testing, security testing, performance testing, and load testing.
 - o **Unit testing:**
 - Unit testing should focus on testing the specific unit/business logic in the code and mocking all the external calls.
 - We can use tools like xUnit, NUnit, MSTest to write unit tests.
 - Unit tests should always be automated in build pipelines to ensure that no code gets checked in without proper unit tests.
 - o **Integration testing:**
 - Integration testing verifies end to end flow of the application and validates that the software behaves as expected when all the moving parts are integrated.
 - o **Security testing:**
 - Security testing focus on revealing any threats/vulnerabilities in the software. It should include pen testing, threat modeling, and testing of application/infrastructure for any vulnerabilities.
 - o **Performance testing:**
 - Performance testing helps in benchmarking the SLAs/response time for software.
 - This testing works in tandem with load/stress testing and is used to identify application boundaries.
- While writing unit tests for async methods, always ensure the call to the async method is awaited. If it isn't awaited test will probably always pass.

Debugging

- Use *F10* or *F11* key to start debugging and step into the first line of code in the entry point of your program in Visual Studio 2019.
- Use *Ctrl + F5* to start your program without debugging.
- While developing or debugging your application, ensure that the exception settings are set to break on all managed exceptions. This way, you will get to know the possible exceptions that may be thrown in various scenarios upfront. It helps avoid issues in production:

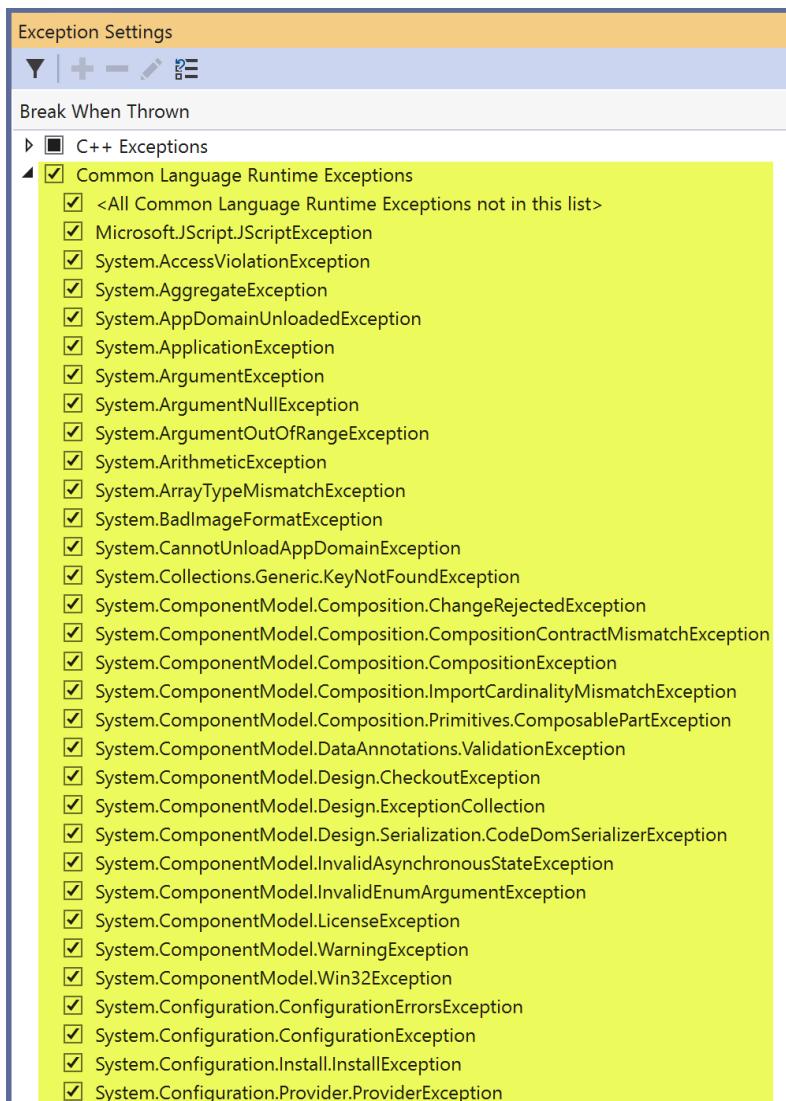


Figure 10.1: Exception settings

- While debugging, do make use of pinning the data tips for better debugging experience, especially in multithreaded scenarios.

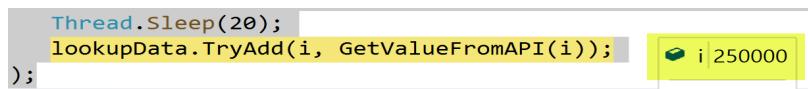


Figure 10.2: Pinned data tips

- For the collaborative development, do make use of **Live Share** feature that is shipped with Visual Studio 2019:

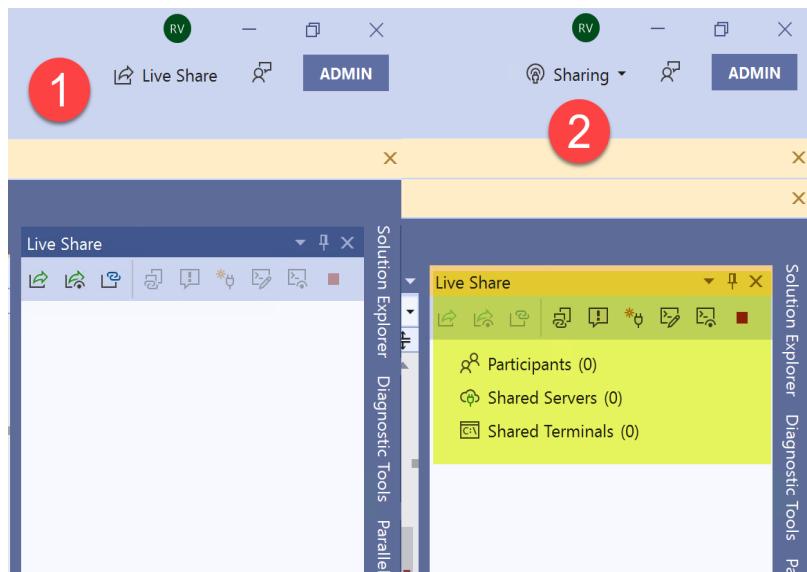


Figure 10.3: Live Share

- Visual Studio helps to focus on the current thread by providing support to focus on the current thread. It can be enabled either by clicking on the highlighted icon below while debugging or by pressing the combination *Ctrl + T*, *Ctrl + T*:



Figure 10.4: Focus on the current thread

- To switch to the next thread, click on the switch to next thread button shown below or by pressing the combination *Ctrl + T*, *Ctrl + J*:



Figure 10.5: Switch to next thread

- Visual Studio has support for (PerfTips) Performance Tips while debugging. This tip appears at the end of the line of code in the code editor window and displays the time taken in the execution of that step or code since the last breakpoint. Screenshot of PerfTip is shown as follows:

```
var stream = assembly.GetManifestResourceStream(resource); ≤ 2ms elapsed
```

Figure 10.6: PerfTip displaying <= 2ms elapsed

- Make use of PerfTips in Visual Studio while debugging your application, to get a real picture of how much time your method calls, and statements are taking and identify expensive calls. These expensive calls should be the focus area for improvement and optimization.
- Utilize the **Diagnostic Tools** window to monitor CPU, memory, events, and exceptions while debugging:

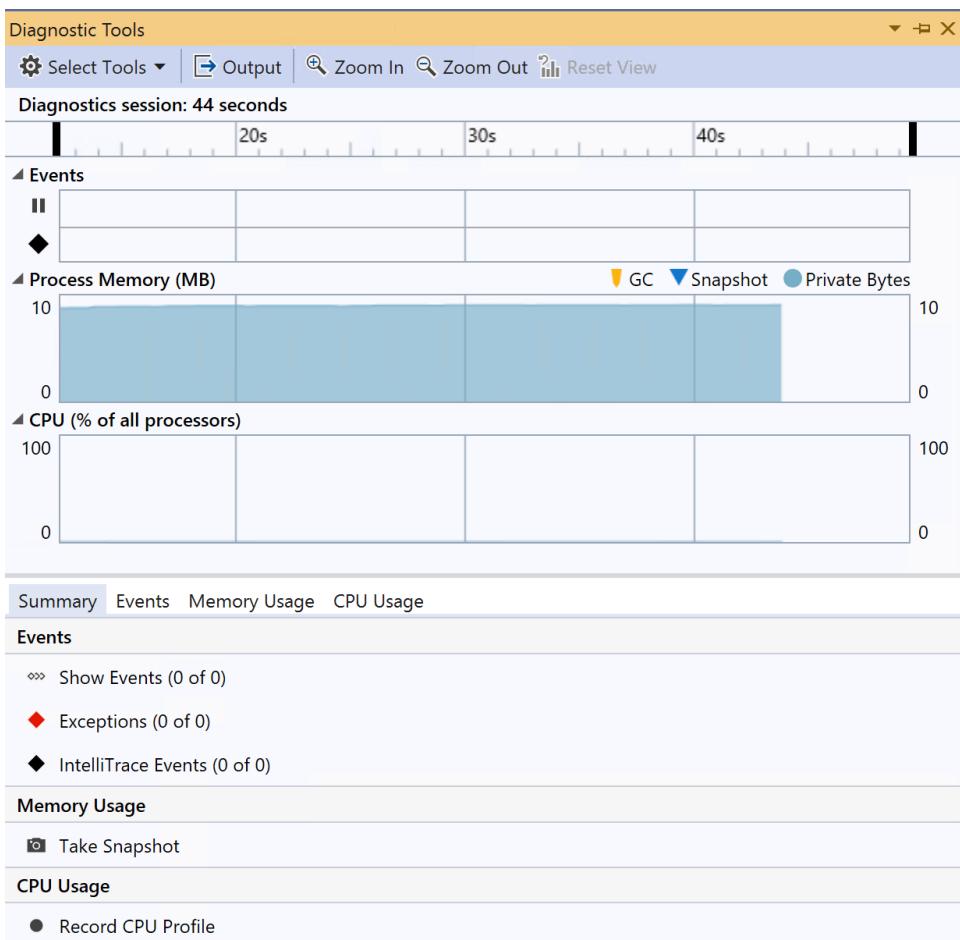
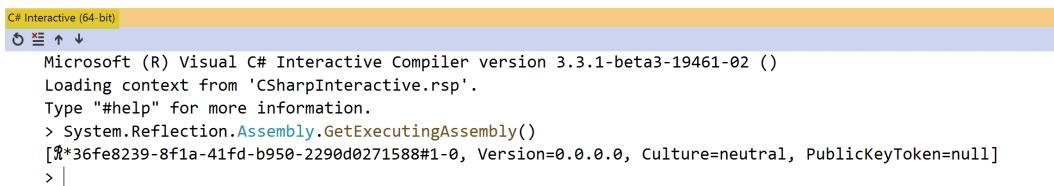


Figure 10.7: Diagnostic Tools

- Do profile your application before shipping it for testing or to the customer. Profiling will uncover the methods and code using CPU extensively or taking up high memory or a long time to execute.
- Make use of a Parallel watch to watch and identify which thread is modifying an object and what is the current value of the object.
- Visual Studio 2019 has excellent support to profile managed applications. Do leverage it! If you have not used it earlier, just type "profile" in Search/Quick launch (*Ctrl + Q*) and get started.
- Make use of the C# Interactive window. It has decent **Read Eval Print Loop (REPL)** support. So, you can execute and test small scripts directly in C# Interactive window:



The screenshot shows the C# Interactive window with the title bar 'C# Interactive (64-bit)'. The window contains the following text:

```

Microsoft (R) Visual C# Interactive Compiler version 3.3.1-beta3-19461-02 ()
Loading context from 'CSharpInteractive.rsp'.
Type "#help" for more information.
> System.Reflection.Assembly.GetExecutingAssembly()
[&#36fe8239-8f1a-41fd-b950-2290d0271588#1-0, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null]
> |

```

Figure 10.8: C# Interactive Window

- For investigating performance issues in the production environment or otherwise, consider using PerfView to collect traces if feasible.
- View out of scope objects by making use of **Make Object ID**. To view the value of out of scope objects, first, add them to the watch window. Then right-click on the object and click on **Make Object ID**. It would create an object id followed by \$ sign. Use this ID to view the values of out of scope objects. Handy! Isn't it!

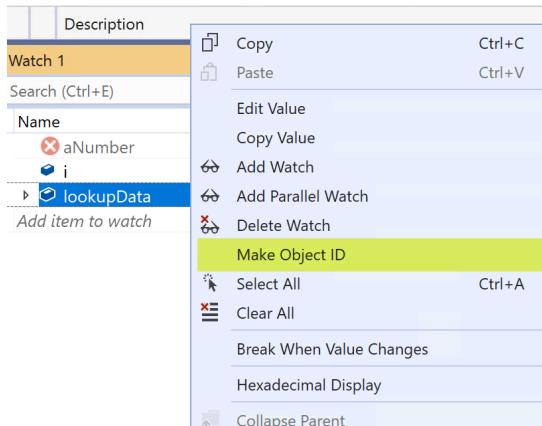
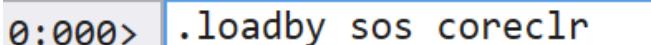


Figure 10.9: Make Object ID

- In **Production**, if you run into issues like high CPU or high memory or deadlock, or hang, or slowness, or crash, collecting memory dump at the time issue occurs may help us investigate and troubleshoot an issue.
- Memory dumps can be collected by several tools, like TaskManager, ProcessExplorer, ProcDump, DebugDiag, and many more, to name a few.
- Memory dumps can be analyzed by several tools as well, like Visual Studio, DebugDiag, WinDbg.
- Bitness of a debugger (x86 or x64) to be used is very important. You may not be able to analyze the dump with the debugger of incorrect bitness.
- While analyzing the memory dump, we may want to see the code of faulting assembly. This tip outlines the steps to do just that. It will make your debugging experience better:

- Load the memory dump in WinDbg
- Load the SOS extension by executing the command:

```
.loadby sos coreclr
```



```
0:000> .loadby sos coreclr
```

Figure 10.10: load SOS

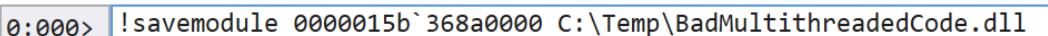
- List all the loaded modules by executing the command
lmv
- Then identify the module for which you wish to view the code.



```
0:000> lmv
start          end            module name
0000015b`368a0000 0000015b`368a0000  BadMultithreadedCode  (deferred)
Image path: C:\Users\rishabhv\source\repos\Parallel-Programming-with-C-8-and-.NET-Core-3.0\Chapter9\BadMultithreadedCode\bin\Debug\netcoreapp3.0\BadMultithreadedCode.dll
Image name: BadMultithreadedCode.dll
Browse all global symbols functions data
Has CLR image header, track-debug-data flag not set
Image was built with /BPREPRO flag.
Timestamp:    A0E6CFD9 (This is a reproducible build file hash, not a timestamp)
Cpu:          00000000
ImageSize:    00000000
FileVersion:  1.0.0.0
ProductVersion: 1.0.0.0
FileFlags:    0 (Mask 3F)
FileOS:       4 Unknown Win32
FileType:     1.0 App
FileDate:    00000000.00000000
Translations: 0000.0409 0000.04e4 0409.04b0 0409.04e4
Information from resource tables:
```

Figure 10.11: Identify the module

- Save the assembly by executing the command:
!SaveModuleby specifying the address found from the start address of the module and path where assembly needs to be saved:



```
0:000> !savemodule 0000015b`368a0000 c:\Temp\BadMultithreadedCode.dll
```

Figure 10.12: Save the module

- o Using dotPeek or justDecompile, generate the pdb of this assembly.

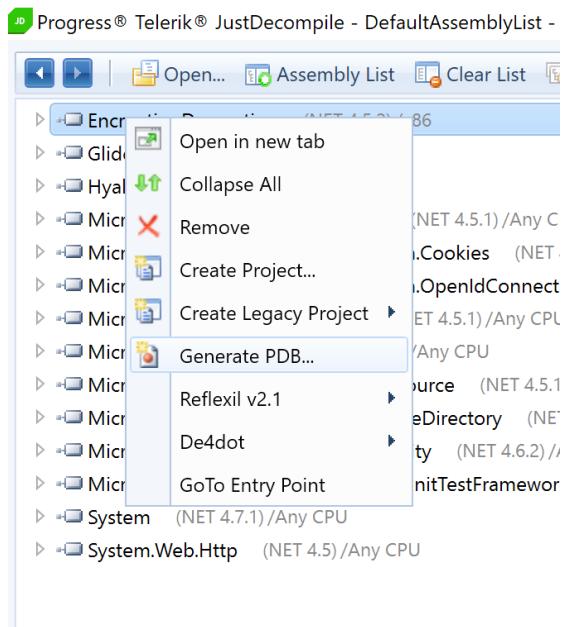


Figure 10.13: Generate PDB

- o Either see the code directly from generated code from these tools or, open the file from WinDbg, by executing the command:

```
.open -a <<address>>
```

Specifying the address of the module. The screenshots shown in *figure 10.11* and *10.12* demonstrate the use of the address.

Azure

- **Leverage Application Insights** to track page views, events, exceptions, dependencies, custom events, requests, and so on.

- Configure auto-scaling rules to handle the burst of a high number of requests.

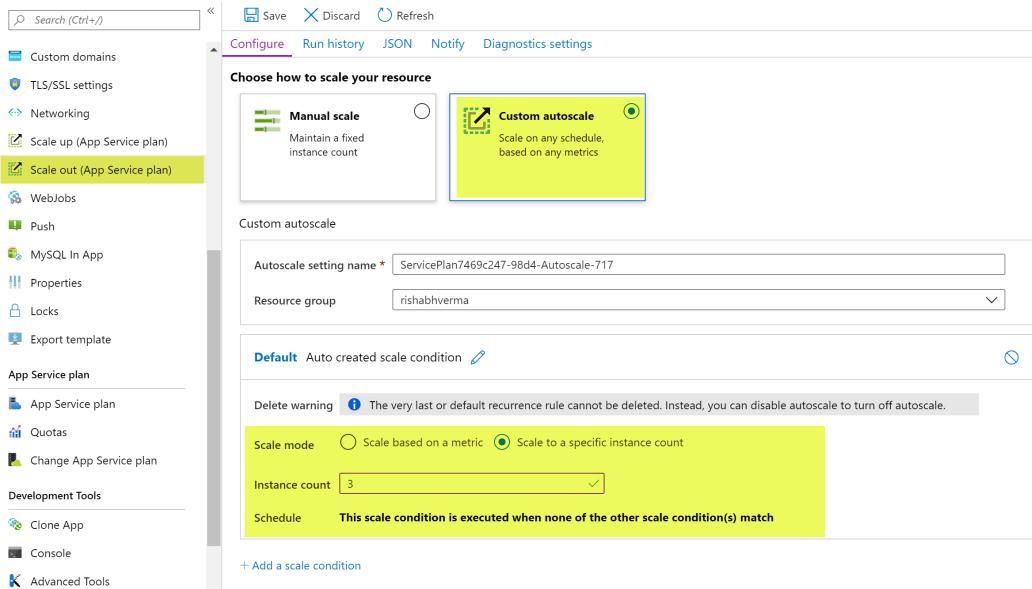


Figure 10.14: Auto Scale-out Configuration

- Use Azure Monitor to monitor your app:

Figure 10.15: Azure Monitor

- Use **Performance Tests** support provided by Azure to load test your applications or APIs:

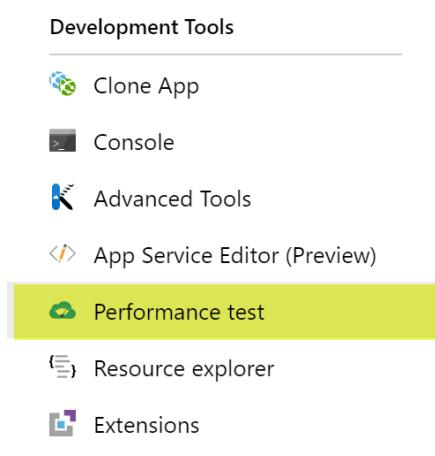


Figure 10.16: Performance Test in App Service

- To enable logging in your web app, to identify issues quickly if they happen:

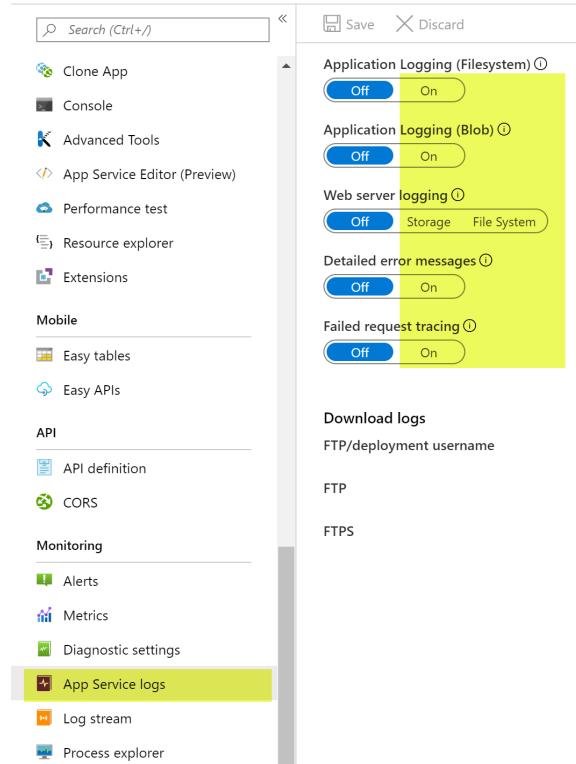


Figure 10.17: App Service logging

- Do leverage **Application Insights web** and availability tests to keep track of the availability of your web application.
- Ensure that all the connections to Azure storage or SQL are adequately disposed of or closed after the calls are made and do make use of connection pooling.
- Do regular health checks of your app to find any issues. ASP.NET Core 3.1 ships with excellent support to include health checks in your application.
- Make use of **Diagnose and solve problems** in your app service to identify and troubleshoot issues with your App Service. Based on the problem to troubleshoot, navigate to the appropriate section, as demonstrated in the next image. The keywords of issue types are highlighted:

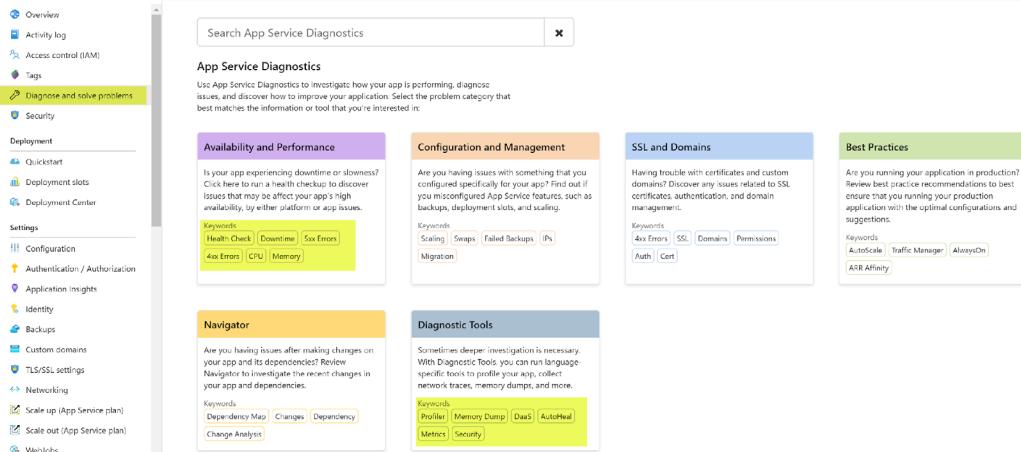


Figure 10.18: Diagnose and solve the problem

Summary

In this chapter, we discussed some neat and handy tips to do better multithreaded and parallel programming.