



Quick answers to common problems

Multithreading with C# Cookbook

Second Edition

Over 70 recipes to get you writing powerful and efficient
multithreaded, asynchronous, and parallel programs in C# 6.0

Eugene Agafonov

[PACKT] open source*
PUBLISHING community experience distilled

Multithreading with C# Cookbook

Second Edition

Over 70 recipes to get you writing powerful
and efficient multithreaded, asynchronous,
and parallel programs in C# 6.0

Eugene Agafonov

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Multithreading with C# Cookbook

Second Edition

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2013

Second Edition: April 2016

Production reference: 1150416

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-125-1

www.packtpub.com

Credits

Author

Eugene Agafonov

Copy Editor

Neha Vyas

Reviewers

Chad McCallum

Philip Pierce

Project Coordinator

Francina Pinto

Commissioning Editor

Edward Gordon

Proofreader

Safis Editing

Acquisition Editor

Kirk D'Costa

Indexer

Rekha Nair

Content Development Editor

Nikhil Borkar

Production Coordinator

Manu Joseph

Technical Editor

Vivek Pala

Cover Work

Manu Joseph

About the Author

Eugene Agafonov leads the development department at ABBYY and lives in Moscow. He has over 15 years of professional experience in software development, and he started working with C# when it was in beta version. He is a Microsoft MVP in ASP.NET since 2006, and he often speaks at local software development conferences, such as DevCon Russia, about cutting-edge technologies in modern web and server-side application development. His main professional interests are cloud-based software architecture, scalability, and reliability. Eugene is a huge fan of football and plays the guitar with a local rock band. You can reach him at his personal blog, eugeneagafonov.com, or find him on Twitter at [@eugene_agafonov](https://twitter.com/eugene_agafonov).

ABBYY is a global leader in the development of document recognition, content capture, and language-based technologies and solutions that are integrated across the entire information life cycle.

He is the author of *Multithreading in C# 5.0 Cookbook* and *Mastering C# Concurrency* by Packt Publishing.

I'd like to dedicate this book to my dearly beloved wife, Helen, and son, Nikita.

About the Reviewers

Chad McCallum is a Saskatchewan computer geek with a passion for software development. He has over 10 years of .NET experience (and 2 years of PHP, but we won't talk about that). After graduating from SIAST Kelsey Campus, he picked up freelance PHP contracting work until he could pester iQmetrix to give him a job, which he's hung onto for the last 10 years. He's come back to his roots in Regina and started HackREGINA, a local hackathon organization aimed at strengthening the developer community while coding and drinking beer. His current focus is mastering the art of multitenant e-commerce with .NET. Between his obsession with board gaming and random app ideas, he tries to learn a new technology every week. You can see the results at www.rttigger.com.

Philip Pierce is a software developer with 20 years of experience in mobile, web, desktop, and server development, database design and management, and game development. His background includes creating A.I. for games and business software, converting AAA games between various platforms, developing multithreaded applications, and creating patented client/server communication technologies.

Philip has won several hackathons, including Best Mobile App at the AT&T Developer Summit 2013, and a runner up for Best Windows 8 App at PayPal's Battlethon Miami. His most recent project was converting Rail Rush and Temple Run 2 from the Android platform to Arcade platforms.

Philip's portfolios can be found at the following websites:

- ▶ <http://www.rocketgamesmobile.com>
- ▶ <http://www.philippiercedeveloper.com>

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print, and bookmark content
- ▶ On demand and accessible via a web browser

Table of Contents

Preface	v
Chapter 1: Threading Basics	1
Introduction	2
Creating a thread in C#	2
Pausing a thread	6
Making a thread wait	7
Aborting a thread	8
Determining a thread state	10
Thread priority	12
Foreground and background threads	14
Passing parameters to a thread	16
Locking with a C# lock keyword	19
Locking with a Monitor construct	22
Handling exceptions	24
Chapter 2: Thread Synchronization	27
Introduction	27
Performing basic atomic operations	28
Using the Mutex construct	31
Using the SemaphoreSlim construct	32
Using the AutoResetEvent construct	34
Using the ManualResetEventSlim construct	36
Using the CountdownEvent construct	38
Using the Barrier construct	39
Using the ReaderWriterLockSlim construct	41
Using the SpinWait construct	44

Chapter 3: Using a Thread Pool	47
Introduction	47
Invoking a delegate on a thread pool	49
Posting an asynchronous operation on a thread pool	52
A thread pool and the degree of parallelism	54
Implementing a cancellation option	56
Using a wait handle and timeout with a thread pool	59
Using a timer	61
Using the BackgroundWorker component	63
Chapter 4: Using the Task Parallel Library	67
Introduction	67
Creating a task	69
Performing basic operations with a task	70
Combining tasks	72
Converting the APM pattern to tasks	75
Converting the EAP pattern to tasks	79
Implementing a cancelation option	81
Handling exceptions in tasks	83
Running tasks in parallel	85
Tweaking the execution of tasks with TaskScheduler	87
Chapter 5: Using C# 6.0	93
Introduction	93
Using the await operator to get asynchronous task results	96
Using the await operator in a lambda expression	98
Using the await operator with consequent asynchronous tasks	100
Using the await operator for the execution of parallel asynchronous tasks	102
Handling exceptions in asynchronous operations	104
Avoiding the use of the captured synchronization context	107
Working around the async void method	111
Designing a custom awaitable type	114
Using the dynamic type with await	118
Chapter 6: Using Concurrent Collections	123
Introduction	123
Using ConcurrentDictionary	125
Implementing asynchronous processing using ConcurrentQueue	127
Changing asynchronous processing order with ConcurrentStack	130
Creating a scalable crawler with ConcurrentBag	132
Generalizing asynchronous processing with BlockingCollection	136

Chapter 7: Using PLINQ	141
Introduction	141
Using the Parallel class	143
Parallelizing a LINQ query	145
Tweaking the parameters of a PLINQ query	148
Handling exceptions in a PLINQ query	151
Managing data partitioning in a PLINQ query	153
Creating a custom aggregator for a PLINQ query	157
Chapter 8: Reactive Extensions	161
Introduction	161
Converting a collection to an asynchronous Observable	162
Writing custom Observable	165
Using the Subjects type	168
Creating an Observable object	172
Using LINQ queries against an observable collection	174
Creating asynchronous operations with Rx	177
Chapter 9: Using Asynchronous I/O	181
Introduction	181
Working with files asynchronously	183
Writing an asynchronous HTTP server and client	187
Working with a database asynchronously	190
Calling a WCF service asynchronously	194
Chapter 10: Parallel Programming Patterns	199
Introduction	199
Implementing Lazy-evaluated shared states	200
Implementing Parallel Pipeline with BlockingCollection	205
Implementing Parallel Pipeline with TPL DataFlow	210
Implementing Map/Reduce with PLINQ	215
Chapter 11: There's More	221
Introduction	221
Using a timer in a Universal Windows Platform application	223
Using WinRT from usual applications	227
Using BackgroundTask in Universal Windows Platform applications	230
Running a .NET Core application on OS X	237
Running a .NET Core application on Ubuntu Linux	240
Index	243

Preface

Not so long ago, a typical personal computer CPU had only one computing core, and the power consumption was enough to cook fried eggs on it. In 2005, Intel introduced its first multiple-core CPU, and since then, computers started developing in a different direction. Low-power consumption and a number of computing cores became more important than a row computing core performance. This lead to programming paradigm changes as well. Now, we need to learn how to use all CPU cores effectively to achieve the best performance, and at the same time, we need to save battery power by running only the programs that we need at a particular time. Besides that, we need to program server applications in a way to use multiple CPU cores or even multiple computers as efficiently as possible to support as many users as we can.

To be able to create such applications, you have to learn to use multiple CPU cores in your programs effectively. If you use the Microsoft .NET development platform and C#, this book will be a perfect starting point for you to program fast and responsive applications.

The purpose of this book is to provide you with a step-by-step guide for multithreading and parallel programming in C#. We will start with the basic concepts, going through more and more advanced topics based on the information from previous chapters, and we will end with real-world parallel programming patterns, Universal Windows applications, and cross-platform applications samples.

What this book covers

Chapter 1, Threading Basics, introduces the basic operations with threads in C#. It explains what a thread is, the pros and cons of using threads, and other important thread aspects.

Chapter 2, Thread Synchronization, describes thread interaction details. You will learn why we need to coordinate threads together and the different ways of organizing thread coordination.

Chapter 3, Using a Thread Pool, explains the thread pool concept. It shows how to use a thread pool, how to work with asynchronous operations, and the good and bad practices of using a thread pool.

Chapter 4, Using the Task Parallel Library, is a deep dive into the Task Parallel Library (TPL) framework. This chapter outlines every important aspect of TPL, including task combination, exception management, and operation cancelation.

Chapter 5, Using C# 6.0, explains in detail the recently introduced C# feature—asynchronous methods. You will find out what the `async` and `await` keywords mean, how to use them in different scenarios, and how `await` works under the hood.

Chapter 6, Using Concurrent Collections, describes the standard data structures for parallel algorithms included in .NET Framework. It goes through sample programming scenarios for each data structure.

Chapter 7, Using PLINQ, is a deep dive into the Parallel LINQ infrastructure. The chapter describes task and data parallelism, parallelizing a LINQ query, tweaking parallelism options, partitioning a query, and aggregating the parallel query result.

Chapter 8, Reactive Extensions, explains how and when to use the Reactive Extensions framework. You will learn how to compose events and how to perform a LINQ query against an event sequence.

Chapter 9, Using Asynchronous I/O, covers in detail the asynchronous I/O process, including files, networks, and database scenarios.

Chapter 10, Parallel Programming Patterns, outlines the solutions to common parallel programming problems.

Chapter 11, There's More, covers the aspects of programming asynchronous applications for Windows 10, OS X, and Linux. You will learn how to work with Windows 10 asynchronous APIs and how to perform the background work in Universal Windows applications. Also, you will get familiar with cross-platform .NET development tools and components.

What you need for this book

For most of the recipes, you will need Microsoft Visual Studio Community 2015. The recipes in *Chapter 11, There's more*, for OS X and Linux will optionally require the Visual Studio Code editor. However, you can use any specific editor you are familiar with.

Who this book is for

This book is written for existing C# developers with little or no background in multithreading and asynchronous and parallel programming. The book covers these topics from basic concepts to complicated programming patterns and algorithms using the C# and .NET ecosystem.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "When the program is run, it creates a thread that will execute a code in the `PrintNumbersWithDelay` method."

A block of code is set as follows:

```
static void LockTooMuch(object lock1, object lock2)
{
    lock (lock1)
    {
        Sleep(1000);
        lock (lock2);
    }
}
```

Any command-line input or output is written as follows:

```
dotnet restore
```

```
dotnet run
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Right-click on the **References** folder in the project, and select the **Manage NuGet Packages...** menu option".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- ▶ WinRAR / 7-Zip for Windows
- ▶ Zipeg / iZip / UnRarX for Mac
- ▶ 7-Zip / PeaZip for Linux

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Threading Basics

In this chapter, we will cover the basic tasks to work with threads in C#. You will learn the following recipes:

- ▶ Creating a thread in C#
- ▶ Pausing a thread
- ▶ Making a thread wait
- ▶ Aborting a thread
- ▶ Determining a thread state
- ▶ Thread priority
- ▶ Foreground and background threads
- ▶ Passing parameters to a thread
- ▶ Locking with a C# lock keyword
- ▶ Locking with a `Monitor` construct
- ▶ Handling exceptions

Introduction

At some point of time in the past, the common computer had only one computing unit and could not execute several computing tasks simultaneously. However, operating systems could already work with multiple programs simultaneously, implementing the concept of multitasking. To prevent the possibility of one program taking control of the CPU forever, causing other applications and the operating system itself to hang, the operating systems had to split a physical computing unit across a few virtualized processors in some way and give a certain amount of computing power to each executing program. Moreover, an operating system must always have priority access to the CPU and should be able to prioritize CPU access to different programs. A thread is an implementation of this concept. It could be considered as a virtual processor that is given to the one specific program and runs it independently.



Remember that a thread consumes a significant amount of operating system resources. Trying to share one physical processor across many threads will lead to a situation where an operating system is busy just managing threads instead of running programs.

Therefore, while it was possible to enhance computer processors, making them execute more and more commands per second, working with threads was usually an operating system task. There was no sense in trying to compute some tasks in parallel on a single-core CPU because it would take more time than running those computations sequentially. However, when processors started to have more computing cores, older programs could not take advantage of this because they just used one processor core.

To use a modern processor's computing power effectively, it is very important to be able to compose a program in a way that it can use more than one computing core, which leads to organizing it as several threads that communicate and synchronize with each other.

The recipes in this chapter focus on performing some very basic operations with threads in the C# language. We will cover a thread's life cycle, which includes creating, suspending, making a thread wait, and aborting a thread, and then, we will go through the basic synchronization techniques.

Creating a thread in C#

Throughout the following recipes, we will use Visual Studio 2015 as the main tool to write multithreaded programs in C#. This recipe will show you how to create a new C# program and use threads in it.



A free Visual Studio Community 2015 IDE can be downloaded from the Microsoft website and used to run the code samples.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found in the `BookSamples\Chapter1\Recipe1` directory.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:



- ▶ Log in or register to our website using your e-mail address and password.
- ▶ Hover the mouse pointer on the **SUPPORT** tab at the top.
- ▶ Click on **Code Downloads & Errata**.
- ▶ Enter the name of the book in the **Search** box.
- ▶ Select the book for which you're looking to download the code files.
- ▶ Choose from the drop-down menu where you purchased this book from.
- ▶ Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

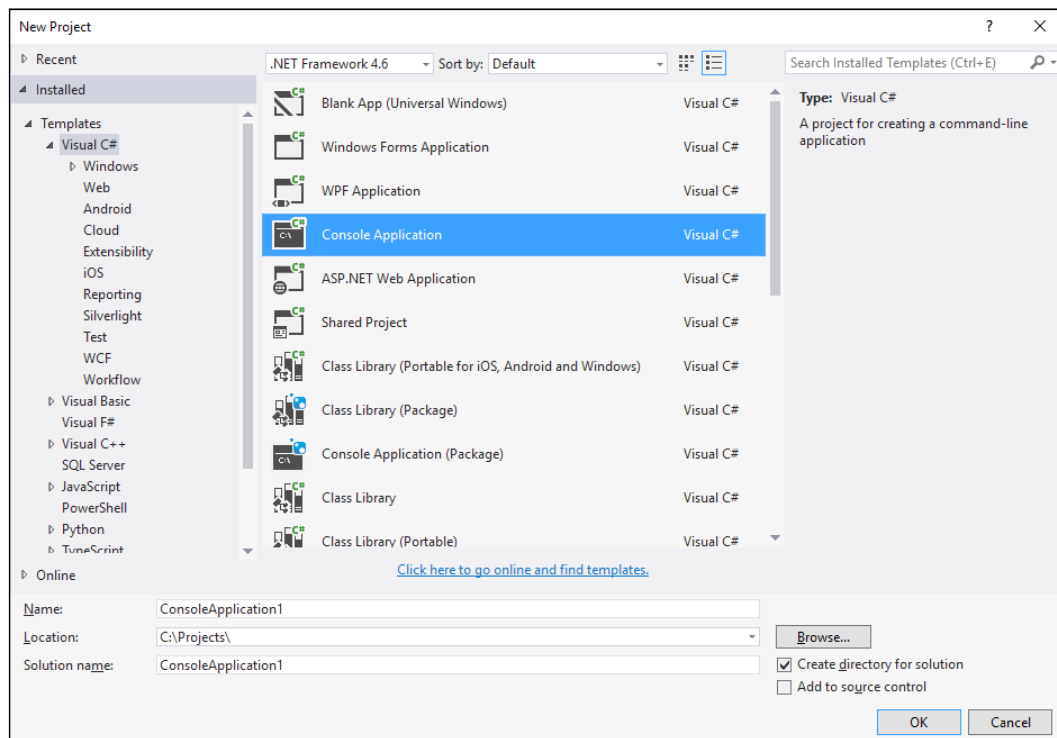
- ▶ WinRAR/7-Zip for Windows
- ▶ Zipeg/iZip / UnRarX for Mac
- ▶ 7-Zip/PeaZip for Linux

How to do it...

To understand how to create a new C# program and use threads in it, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.

2. Make sure that the project uses .NET Framework 4.6 or higher; however, the code in this chapter will work with previous versions.



3. In the Program.cs file, add the following using directives:

```
using System;
using System.Threading;
using static System.Console;
```

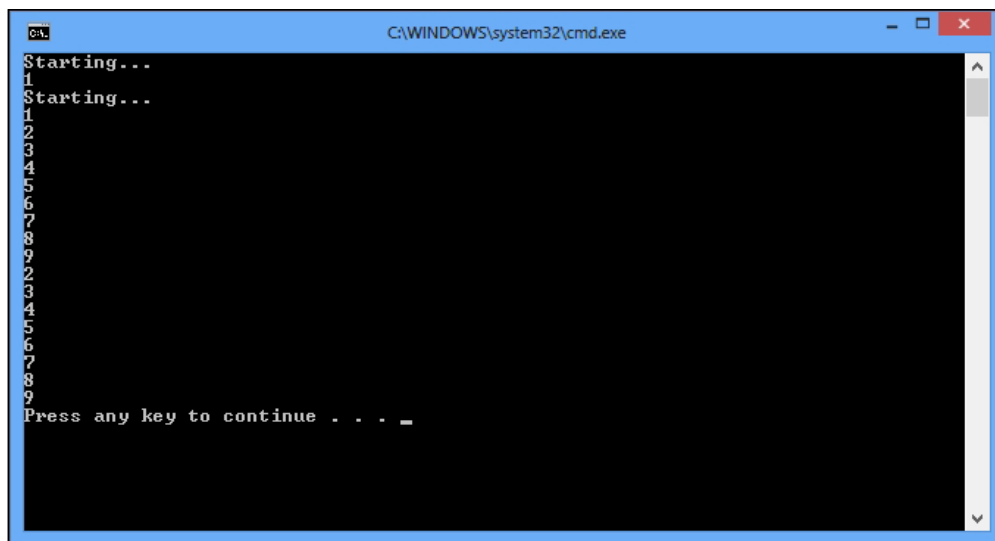
4. Add the following code snippet below the Main method:

```
static void PrintNumbers()
{
    WriteLine("Starting...");
    for (int i = 1; i < 10; i++)
    {
        WriteLine(i);
    }
}
```

5. Add the following code snippet inside the `Main` method:

```
Thread t = new Thread(PrintNumbers);  
t.Start();  
PrintNumbers();
```

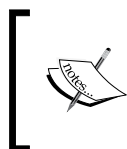
6. Run the program. The output will be something like the following screenshot:



```
C:\WINDOWS\system32\cmd.exe  
Starting...  
1  
Starting...  
1  
2  
3  
4  
5  
6  
7  
8  
9  
2  
3  
4  
5  
6  
7  
8  
9  
Press any key to continue . . . _
```

How it works...

In step 1 and 2, we created a simple console application in C# using .Net Framework version 4.0. Then, in step 3, we included the `System.Threading` namespace, which contains all the types needed for the program. Then, we used the `using static` feature from C# 6.0, which allows us to use the `System.Console` type's static methods without specifying the type name.



An instance of a program that is being executed can be referred to as a process. A process consists of one or more threads. This means that when we run a program, we always have one main thread that executes the program code.

In step 4, we defined the `PrintNumbers` method, which will be used in both the main and newly created threads. Then, in step 5, we created a thread that runs `PrintNumbers`. When we construct a thread, an instance of the `ThreadStart` or `ParameterizedThreadStart` delegate is passed to the constructor. The C# compiler creates this object behind the scenes when we just type the name of the method we want to run in a different thread. Then, we start a thread and run `PrintNumbers` in the usual manner on the main thread.

As a result, there will be two ranges of numbers from 1 to 10 randomly crossing each other. This illustrates that the `PrintNumbers` method runs simultaneously on the main thread and on the other thread.

Pausing a thread

This recipe will show you how to make a thread wait for some time without wasting operating system resources.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe2`.

How to do it...

To understand how to make a thread wait without wasting operating system resources, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void PrintNumbers()
{
    WriteLine("Starting...");
    for (int i = 1; i < 10; i++)
    {
        WriteLine(i);
    }
}

static void PrintNumbersWithDelay()
{
    WriteLine("Starting...");
    for (int i = 1; i < 10; i++)
    {
        Sleep(TimeSpan.FromSeconds(2));
    }
}
```

```
        WriteLine(i);  
    }  
}
```

4. Add the following code snippet inside the `Main` method:

```
Thread t = new Thread(PrintNumbersWithDelay);  
t.Start();  
PrintNumbers();
```

5. Run the program.

How it works...

When the program is run, it creates a thread that will execute a code in the `PrintNumbersWithDelay` method. Immediately after that, it runs the `PrintNumbers` method. The key feature here is adding the `Thread.Sleep` method call to a `PrintNumbersWithDelay` method. It causes the thread executing this code to wait a specified amount of time (2 seconds in our case) before printing each number. While a thread sleeps, it uses as little CPU time as possible. As a result, we will see that the code in the `PrintNumbers` method, which usually runs later, will be executed before the code in the `PrintNumbersWithDelay` method in a separate thread.

Making a thread wait

This recipe will show you how a program can wait for some computation in another thread to complete to use its result later in the code. It is not enough to use the `Thread.Sleep` method because we don't know the exact time the computation will take.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe3`.

How to do it...

To understand how a program waits for some computation in another thread to complete in order to use its result later, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;  
using System.Threading;
```



```
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the Main method:

```
static void PrintNumbersWithDelay()
{
    WriteLine("Starting...");
    for (int i = 1; i < 10; i++)
    {
        Sleep(TimeSpan.FromSeconds(2));
        WriteLine(i);
    }
}
```

4. Add the following code snippet inside the Main method:

```
WriteLine("Starting...");
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
t.Join();
WriteLine("Thread completed");
```

5. Run the program.

How it works...

When the program is run, it runs a long-running thread that prints out numbers and waits two seconds before printing each number. But, in the main program, we called the `t.Join` method, which allows us to wait for the thread `t` to complete working. When it is complete, the main program continues to run. With the help of this technique, it is possible to synchronize execution steps between two threads. The first one waits until another one is complete and then continues to work. While the first thread waits, it is in a `blocked` state (as it is in the previous recipe when you call `Thread.Sleep`).

Aborting a thread

In this recipe, we will describe how to abort another thread's execution.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe4`.

How to do it...

To understand how to abort another thread's execution, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:
3. Using the static `System.Threading.Thread`, add the following code snippet below the `Main` method:

```
using System;
using System.Threading;
using static System.Console;

static void PrintNumbersWithDelay()
{
    WriteLine("Starting...");
    for (int i = 1; i < 10; i++)
    {
        Sleep(TimeSpan.FromSeconds(2));
        WriteLine(i);
    }
}
```

4. Add the following code snippet inside the `Main` method:

```
WriteLine("Starting program...");
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
Thread.Sleep(TimeSpan.FromSeconds(6));
t.Abort();
WriteLine("A thread has been aborted");
Thread t = new Thread(PrintNumbers);
t.Start();
PrintNumbers();
```

5. Run the program.

How it works...

When the main program and a separate number-printing thread run, we wait for six seconds and then call a `t.Abort` method on a thread. This injects a `ThreadAbortException` method into a thread, causing it to terminate. It is very dangerous, generally because this exception can happen at any point and may totally destroy the application. In addition, it is not always possible to terminate a thread with this technique. The target thread may refuse to abort by handling this exception by calling the `Thread.ResetAbort` method. Thus, it is not recommended that you use the `Abort` method to close a thread. There are different methods that are preferred, such as providing a `CancellationToken` object to cancel a thread execution. This approach will be described in *Chapter 3, Using a Thread Pool*.

Determining a thread state

This recipe will describe the possible states a thread could have. It is useful to get information about whether a thread is started yet or whether it is in a blocked state. Note that because a thread runs independently, its state could be changed at any time.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe5`.

How to do it...

To understand how to determine a thread state and acquire useful information about it, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void DoNothing()
{
    Sleep(TimeSpan.FromSeconds(2));
}

static void PrintNumbersWithStatus()
```

```

{
    WriteLine("Starting...");
    WriteLine(CurrentThread.ThreadState.ToString());
    for (int i = 1; i < 10; i++)
    {
        Sleep(TimeSpan.FromSeconds(2));
        WriteLine(i);
    }
}

```

4. Add the following code snippet inside the Main method:

```

WriteLine("Starting program...");
Thread t = new Thread(PrintNumbersWithStatus);
Thread t2 = new Thread(DoNothing);
WriteLine(t.ThreadState.ToString());
t2.Start();
t.Start();
for (int i = 1; i < 30; i++)
{
    WriteLine(t.ThreadState.ToString());
}
Sleep(TimeSpan.FromSeconds(6));
t.Abort();
WriteLine("A thread has been aborted");
WriteLine(t.ThreadState.ToString());
WriteLine(t2.ThreadState.ToString());

```

5. Run the program.

How it works...

When the main program starts, it defines two different threads; one of them will be aborted and the other runs successfully. The thread state is located in the `ThreadState` property of a `Thread` object, which is a C# enumeration. At first, the thread has a `ThreadState.Unstarted` state. Then, we run it and assume that for the duration of 30 iterations of a cycle, the thread will change its state from `ThreadState.Running` to `ThreadState.WaitSleepJoin`.



Note that the current `Thread` object is always accessible through the `Thread.CurrentThread` static property.

If this does not happen, just increase the number of iterations. Then, we abort the first thread and see that now it has a `ThreadState.Aborted` state. It is also possible that the program will print out the `ThreadState.AbortRequested` state. This illustrates, very well, the complexity of synchronizing two threads. Keep in mind that you should not use thread abortion in your programs. I've covered it here only to show the corresponding thread state.

Finally, we can see that our second thread `t2` was completed successfully and now has a `ThreadState.Stopped` state. There are several other states, but they are partly deprecated and not as useful as those we examined.

Thread priority

This recipe will describe the different options for thread priority. Setting a thread priority determines how much CPU time a thread will be given.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe6`.

How to do it...

To understand the workings of thread priority, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
using static System.Diagnostics.Process;
```

3. Add the following code snippet below the `Main` method:

```
static void RunThreads()
{
    var sample = new ThreadSample();

    var threadOne = new Thread(sample.CountNumbers);
    threadOne.Name = "ThreadOne";
    var threadTwo = new Thread(sample.CountNumbers);
    threadTwo.Name = "ThreadTwo";

    threadOne.Priority = ThreadPriority.Highest;
```

```

        threadTwo.Priority = ThreadPriority.Lowest;
        threadOne.Start();
        threadTwo.Start();

        Sleep(TimeSpan.FromSeconds(2));
        sample.Stop();
    }

    class ThreadSample
    {
        private bool _isStopped = false;

        public void Stop()
        {
            _isStopped = true;
        }

        public void CountNumbers()
        {
            long counter = 0;

            while (!_isStopped)
            {
                counter++;
            }

            WriteLine($"{CurrentThread.Name} with " +
                $"{CurrentThread.Priority,11} priority " +
                $"has a count = {counter,13:N0}");
        }
    }

```

4. Add the following code snippet inside the Main method:

```

WriteLine($"Current thread priority: {CurrentThread.Priority}");
WriteLine("Running on all cores available");
RunThreads();
Sleep(TimeSpan.FromSeconds(2));
WriteLine("Running on a single core");
GetCurrentProcess().ProcessorAffinity = new IntPtr(1);
RunThreads();

```

5. Run the program.

How it works...

When the main program starts, it defines two different threads. The first one, `threadOne`, has the highest thread priority `ThreadPriority.Highest`, while the second one, that is `threadTwo`, has the lowest `ThreadPriority.Lowest` priority. We print out the main thread priority value and then start these two threads on all available cores. If we have more than one computing core, we should get an initial result within two seconds. The highest priority thread should calculate more iterations usually, but both values should be close. However, if there are any other programs running that load all the CPU cores, the situation could be quite different.

To simulate this situation, we set up the `ProcessorAffinity` option, instructing the operating system to run all our threads on a single CPU core (number 1). Now, the results should be very different, and the calculations will take more than two seconds. This happens because the CPU core runs mostly the high-priority thread, giving the rest of the threads very little time.

Note that this is an illustration of how an operating system works with thread prioritization. Usually, you should not write programs relying on this behavior.

Foreground and background threads

This recipe will describe what foreground and background threads are and how setting this option affects the program's behavior.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe7`.

How to do it...

To understand the effect of foreground and background threads on a program, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the Main method:

```
class ThreadSample
{
    private readonly int _iterations;

    public ThreadSample(int iterations)
    {
        _iterations = iterations;
    }
    public void CountNumbers()
    {
        for (int i = 0; i < _iterations; i++)
        {
            Sleep(TimeSpan.FromSeconds(0.5));
            WriteLine($"{CurrentThread.Name} prints {i}");
        }
    }
}
```

4. Add the following code snippet inside the Main method:

```
var sampleForeground = new ThreadSample(10);
var sampleBackground = new ThreadSample(20);

var threadOne = new Thread(sampleForeground.CountNumbers);
threadOne.Name = "ForegroundThread";
var threadTwo = new Thread(sampleBackground.CountNumbers);
threadTwo.Name = "BackgroundThread";
threadTwo.IsBackground = true;

threadOne.Start();
threadTwo.Start();
```

5. Run the program.

How it works...

When the main program starts, it defines two different threads. By default, a thread that we create explicitly is a foreground thread. To create a background thread, we manually set the `IsBackground` property of the `threadTwo` object to `true`. We configure these threads in a way that the first one will be completed faster, and then we run the program.

After the first thread is complete, the program shuts down and the background thread is terminated. This is the main difference between the two: a process waits for all the foreground threads to complete before finishing the work, but if it has background threads, they just shut down.

It is also important to mention that if a program defines a foreground thread that does not get completed; the main program does not end properly.

Passing parameters to a thread

This recipe will describe how to provide code that we run in another thread with the required data. We will go through the different ways to fulfill this task and review common mistakes.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe8`.

How to do it...

To understand how to pass parameters to a thread, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void Count(object iterations)
{
    CountNumbers((int)iterations);
}

static void CountNumbers(int iterations)
{
    for (int i = 1; i <= iterations; i++)
    {
        Sleep(TimeSpan.FromSeconds(0.5));
        WriteLine($"{CurrentThread.Name} prints {i}");
    }
}
```

```

    }

    static void PrintNumber(int number)
    {
        WriteLine(number);
    }

    class ThreadSample
    {
        private readonly int _iterations;

        public ThreadSample(int iterations)
        {
            _iterations = iterations;
        }

        public void CountNumbers()
        {
            for (int i = 1; i <= _iterations; i++)
            {
                Sleep(TimeSpan.FromSeconds(0.5));
                WriteLine($"{CurrentThread.Name} prints {i}");
            }
        }
    }
}

```

4. Add the following code snippet inside the Main method:

```

var sample = new ThreadSample(10);

var threadOne = new Thread(sample.CountNumbers);
threadOne.Name = "ThreadOne";
threadOne.Start();
threadOne.Join();

WriteLine("-----");

var threadTwo = new Thread(Count);
threadTwo.Name = "ThreadTwo";
threadTwo.Start(8);
threadTwo.Join();

WriteLine("-----");

var threadThree = new Thread(() => CountNumbers(12));
threadThree.Name = "ThreadThree";

```

```
threadThree.Start();
threadThree.Join();
WriteLine("-----");

int i = 10;
var threadFour = new Thread(() => PrintNumber(i));
i = 20;
var threadFive = new Thread(() => PrintNumber(i));
threadFour.Start();
threadFive.Start();
```

5. Run the program.

How it works...

When the main program starts, it first creates an object of the `ThreadSample` class, providing it with a number of iterations. Then, we start a thread with the object's `CountNumbers` method. This method runs in another thread, but it uses the number 10, which is the value that we passed to the object's constructor. Therefore, we just passed this number of iterations to another thread in the same indirect way.

There's more...

Another way to pass data is to use the `Thread.Start` method by accepting an object that can be passed to another thread. To work this way, a method that we started in another thread must accept one single parameter of the type `object`. This option is illustrated by creating a `threadTwo` thread. We pass 8 as an object to the `Count` method, where it is cast to an `integer` type.

The next option involves the use of lambda expressions. A lambda expression defines a method that does not belong to any class. We create such a method that invokes another method with the arguments needed and start it in another thread. When we start the `threadThree` thread, it prints out 12 numbers, which are exactly the numbers we passed to it via the lambda expression.

The use of lambda expressions involves another C# construct named `closure`. When we use any local variable in a lambda expression, C# generates a class and makes this variable a property of this class. So, actually, we do the same thing as in the `threadOne` thread, but we do not define the class ourselves; the C# compiler does this automatically.

This could lead to several problems; for example, if we use the same variable from several lambdas, they will actually share this variable value. This is illustrated by the previous example where, when we start `threadFour` and `threadFive`, they both print 20 because the variable was changed to hold the value 20 before both threads were started.

Locking with a C# lock keyword

This recipe will describe how to ensure that when one thread uses some resource, another does not simultaneously use it. We will see why this is needed and what the thread safety concept is all about.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe9`.

How to do it...

To understand how to use the C# lock keyword, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
```

3. Add the following code snippet below the `Main` method:

```
static void TestCounter(CounterBase c)
{
    for (int i = 0; i < 100000; i++)
    {
        c.Increment();
        c.Decrement();
    }
}

class Counter : CounterBase
{
    public int Count { get; private set; }

    public override void Increment()
    {
        Count++;
    }

    public override void Decrement()
```

```
    {  
        Count--;  
    }  
}  
  
class CounterWithLock : CounterBase  
{  
    private readonly object _syncRoot = new Object();  
  
    public int Count { get; private set; }  
  
    public override void Increment()  
    {  
        lock (_syncRoot)  
        {  
            Count++;  
        }  
    }  
  
    public override void Decrement()  
    {  
        lock (_syncRoot)  
        {  
            Count--;  
        }  
    }  
}  
  
abstract class CounterBase  
{  
    public abstract void Increment();  
  
    public abstract void Decrement();  
}
```

4. Add the following code snippet inside the Main method:

```
WriteLine("Incorrect counter");  
  
var c = new Counter();  
  
var t1 = new Thread(() => TestCounter(c));  
var t2 = new Thread(() => TestCounter(c));  
var t3 = new Thread(() => TestCounter(c));  
t1.Start();
```

```

t2.Start();
t3.Start();
t1.Join();
t2.Join();
t3.Join();

WriteLine($"Total count: {c.Count}");
WriteLine("-----");

WriteLine("Correct counter");

var c1 = new CounterWithLock();

t1 = new Thread(() => TestCounter(c1));
t2 = new Thread(() => TestCounter(c1));
t3 = new Thread(() => TestCounter(c1));
t1.Start();
t2.Start();
t3.Start();
t1.Join();
t2.Join();
t3.Join();
WriteLine($"Total count: {c1.Count}");

```

5. Run the program.

How it works...

When the main program starts, it first creates an object of the `Counter` class. This class defines a simple counter that can be incremented and decremented. Then, we start three threads that share the same counter instance and perform an increment and decrement in a cycle. This leads to nondeterministic results. If we run the program several times, it will print out several different counter values. It could be 0, but mostly won't be.

This happens because the `Counter` class is not thread-safe. When several threads access the counter at the same time, the first thread gets the counter value 10 and increments it to 11. Then, a second thread gets the value 11 and increments it to 12. The first thread gets the counter value 12, but before a decrement takes place, a second thread gets the counter value 12 as well. Then, the first thread decrements 12 to 11 and saves it into the counter, and the second thread simultaneously does the same. As a result, we have two increments and only one decrement, which is obviously not right. This kind of a situation is called a race condition and is a very common cause of errors in a multithreaded environment.

To make sure that this does not happen, we must ensure that while one thread works with the counter, all other threads wait until the first one finishes the work. We can use the `lock` keyword to achieve this kind of behavior. If we lock an object, all the other threads that require an access to this object will wait in a blocked state until it is unlocked. This could be a serious performance issue and later, in *Chapter 2, Thread Synchronization*, you will learn more about this.

Locking with a Monitor construct

This recipe illustrates another common multithreaded error called a deadlock. Since a deadlock will cause a program to stop working, the first piece in this example is a new `Monitor` construct that allows us to avoid a deadlock. Then, the previously described `lock` keyword is used to get a deadlock.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe10`.

How to do it...

To understand the multithreaded error deadlock, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void LockTooMuch(object lock1, object lock2)
{
    lock (lock1)
    {
        Sleep(1000);
        lock (lock2);
    }
}
```

4. Add the following code snippet inside the `Main` method:

```
object lock1 = new object();
```

```

object lock2 = new object();

new Thread(() => LockTooMuch(lock1, lock2)).Start();

lock (lock2)
{
    Thread.Sleep(1000);
    WriteLine("Monitor.TryEnter allows not to get stuck, returning
false after a specified timeout is elapsed");
    if (Monitor.TryEnter(lock1, TimeSpan.FromSeconds(5)))
    {
        WriteLine("Acquired a protected resource succesfully");
    }
    else
    {
        WriteLine("Timeout acquiring a resource!");
    }
}

new Thread(() => LockTooMuch(lock1, lock2)).Start();

WriteLine("-----");
lock (lock2)
{
    WriteLine("This will be a deadlock!");
    Sleep(1000);
    lock (lock1)
    {
        WriteLine("Acquired a protected resource succesfully");
    }
}

```

5. Run the program.

How it works...

Let's start with the `LockTooMuch` method. In this method, we just lock the first object, wait for a second, and then lock the second object. Then, we start this method in another thread and try to lock the second object and then the first object from the main thread.

If we use the `lock` keyword like in the second part of this demo, there will be a deadlock. The first thread holds a `lock` on the `lock1` object and waits while the `lock2` object gets free; the main thread holds a `lock` on the `lock2` object and waits for the `lock1` object to become free, which will never happen in this situation.

Actually, the `lock` keyword is syntactic sugar for the `Monitor` class usage. If we were to disassemble code with `lock`, we would see that it turns into the following code snippet:

```
bool acquiredLock = false;
try
{
    Monitor.Enter(lockObject, ref acquiredLock);

    // Code that accesses resources that are protected by the lock.

}
finally
{
    if (acquiredLock)
    {
        Monitor.Exit(lockObject);
    }
}
```

Therefore, we can use the `Monitor` class directly; it has the `TryEnter` method, which accepts a timeout parameter and returns `false` if this timeout parameter expires before we can acquire the resource protected by `lock`.

Handling exceptions

This recipe will describe how to handle exceptions in other threads properly. It is very important to always place a `try/catch` block inside the thread because it is not possible to catch an exception outside a thread's code.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter1\Recipe11`.

How to do it...

To understand the handling of exceptions in other threads, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
```

```
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the Main method:

```
static void BadFaultyThread()
{
    WriteLine("Starting a faulty thread...");
    Sleep(TimeSpan.FromSeconds(2));
    throw new Exception("Boom!");
}

static void FaultyThread()
{
    try
    {
        WriteLine("Starting a faulty thread...");
        Sleep(TimeSpan.FromSeconds(1));
        throw new Exception("Boom!");
    }
    catch (Exception ex)
    {
        WriteLine($"Exception handled: {ex.Message}");
    }
}
```

4. Add the following code snippet inside the Main method:

```
var t = new Thread(FaultyThread);
t.Start();
t.Join();

try
{
    t = new Thread(BadFaultyThread);
    t.Start();
}
catch (Exception ex)
{
    WriteLine("We won't get here!");
}
```

5. Run the program.

How it works...

When the main program starts, it defines two threads that will throw an exception. One of these threads handles an exception, while the other does not. You can see that the second exception is not caught by a `try/catch` block around the code that starts the thread. So, if you work with threads directly, the general rule is to not throw an exception from a thread, but to use a `try/catch` block inside a thread code instead.

In the older versions of .NET Framework (1.0 and 1.1), this behavior was different and uncaught exceptions did not force an application shutdown. It is possible to use this policy by adding an application configuration file (such as `app.config`) that contains the following code snippet:

```
<configuration>
  <runtime>
    <legacyUnhandledExceptionPolicy enabled="1" />
  </runtime>
</configuration>
```

2

Thread Synchronization

In this chapter, we will describe some of the common techniques of working with shared resources from multiple threads. You will learn the following recipes:

- ▶ Performing basic atomic operations
- ▶ Using the `Mutex` construct
- ▶ Using the `SemaphoreSlim` construct
- ▶ Using the `AutoResetEvent` construct
- ▶ Using the `ManualResetEventSlim` construct
- ▶ Using the `CountDownEvent` construct
- ▶ Using the `Barrier` construct
- ▶ Using the `ReaderWriterLockSlim` construct
- ▶ Using the `SpinWait` construct

Introduction

As we saw in *Chapter 1, Threading Basics*, it is problematic to use a shared object simultaneously from several threads. However, it is very important to synchronize those threads so that they perform operations on that shared object in a proper sequence. In the *Locking with a C# lock keyword* recipe, we faced a problem called the race condition. The problem occurred because the execution of those multiple threads was not synchronized properly. When one thread performs increment and decrement operations, the other threads must wait for their turn. Organizing threads in such a way is often referred to as **thread synchronization**.

There are several ways to achieve thread synchronization. First, if there is no shared object, there is no need for synchronization at all. Surprisingly, it is very often the case that we can get rid of complex synchronization constructs by just redesigning our program and removing a shared state. If possible, just avoid using a single object from several threads.

If we must have a shared state, the second approach is to use only **atomic** operations. This means that an operation takes a single quantum of time and completes at once, so no other thread can perform another operation until the first operation is complete. Therefore, there is no need to make other threads wait for this operation to complete and there is no need to use locks; this in turn, excludes the deadlock situation.

If this is not possible and the program's logic is more complicated, then we have to use different constructs to coordinate threads. One group of these constructs puts a waiting thread into a `blocked` state. In a `blocked` state, a thread uses as little CPU time as possible. However, this means that it will include at least one so-called **context switch**—the thread scheduler of an operating system will save the waiting thread's state and switch to another thread, restoring its state by turn. This takes a considerable amount of resources; however, if the thread is going to be suspended for a long time, it is good. These kind of constructs are also called **kernel-mode** constructs because only the kernel of an operating system is able to stop a thread from using CPU time.

In case, we have to wait for a short period of time, it is better to simply wait than switch the thread to a `blocked` state. This will save us the context switch at the cost of some wasted CPU time while the thread is waiting. Such constructs are referred to as **user-mode** constructs. They are very lightweight and fast, but they waste a lot of CPU time in case a thread has to wait for long.

To use the best of both worlds, there are **hybrid** constructs; these try to use user-mode waiting first, and then, if a thread waits long enough, it switches to the `blocked` state, saving CPU resources.

In this chapter, we will look through the aspects of thread synchronization. We will cover how to perform atomic operations and how to use the existing synchronization constructs included in .NET Framework.

Performing basic atomic operations

This recipe will show you how to perform basic atomic operations on an object to prevent the race condition without blocking threads.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe1`.

How to do it...

To understand basic atomic operations, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.

2. In the Program.cs file, add the following using directives:

```
using System;
using System.Threading;
using static System.Console;
```

3. Below the Main method, add the following code snippet:

```
static void TestCounter(CounterBase c)
{
    for (int i = 0; i < 100000; i++)
    {
        c.Increment();
        c.Decrement();
    }
}

class Counter : CounterBase
{
    private int _count;

    public int Count => _count;

    public override void Increment()
    {
        _count++;
    }

    public override void Decrement()
    {
        _count--;
    }
}

class CounterNoLock : CounterBase
{
    private int _count;

    public int Count => _count;

    public override void Increment()
    {
        Interlocked.Increment(ref _count);
    }

    public override void Decrement()
    {

```

```
        Interlocked.Decrement(ref _count);
    }
}
```

```
abstract class CounterBase
{
    public abstract void Increment();

    public abstract void Decrement();
}
```

4. Inside the Main method, add the following code snippet:

```
WriteLine("Incorrect counter");

var c = new Counter();

var t1 = new Thread(() => TestCounter(c));
var t2 = new Thread(() => TestCounter(c));
var t3 = new Thread(() => TestCounter(c));
t1.Start();
t2.Start();
t3.Start();
t1.Join();
t2.Join();
t3.Join();

WriteLine($"Total count: {c.Count}");
WriteLine("-----");

WriteLine("Correct counter");

var c1 = new CounterNoLock();

t1 = new Thread(() => TestCounter(c1));
t2 = new Thread(() => TestCounter(c1));
t3 = new Thread(() => TestCounter(c1));
t1.Start();
t2.Start();
t3.Start();
t1.Join();
t2.Join();
t3.Join();

WriteLine($"Total count: {c1.Count}");
```

5. Run the program.

How it works...

When the program runs, it creates three threads that will execute a code in the `TestCounter` method. This method runs a sequence of increment/decrement operations on an object. Initially, the `Counter` object is not thread-safe and we get a race condition here. So, in the first case, a counter value is not deterministic. We could get a zero value; however, if you run the program several times, you will eventually get some incorrect nonzero result.

In *Chapter 1, Threading Basics*, we resolved this problem by locking our object, causing other threads to be blocked while one thread gets the old counter value and then computes and assigns a new value to the counter. However, if we execute this operation in such a way, it cannot be stopped midway, we would achieve the proper result without any locking, and this is possible with the help of the `Interlocked` construct. It provides the `Increment`, `Decrement`, and `Add` atomic methods for basic math, and it helps us to write the `Counter` class without the use of locking.

Using the Mutex construct

This recipe will describe how to synchronize two separate programs using the `Mutex` construct. A `Mutex` construct is a synchronization primitive that grants exclusive access of the shared resource to only one thread.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe2`.

How to do it...

To understand the synchronization of two separate programs using the `Mutex` construct, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;  
using System.Threading;  
using static System.Console;
```

3. Inside the `Main` method, add the following code snippet:

```
const string MutexName = "CSharpThreadingCookbook";  
  
using (var m = new Mutex(false, MutexName))
```



```
{
    if (!m.WaitOne(TimeSpan.FromSeconds(5), false))
    {
        WriteLine("Second instance is running!");
    }
    else
    {
        WriteLine("Running!");
        ReadLine();
        m.ReleaseMutex();
    }
}
```

4. Run the program.

How it works...

When the main program starts, it defines a mutex with a specific name, providing the `initialOwner` flag as `false`. This allows the program to acquire a mutex if it is already created. Then, if no mutex is acquired, the program simply displays **Running** and waits for any key to be pressed in order to release the mutex and exit.

If we start a second copy of the program, it will wait for 5 seconds, trying to acquire the mutex. If we press any key in the first copy of a program, the second one will start the execution. However, if we keep waiting for 5 seconds, the second copy of the program will fail to acquire the mutex.



Note that a mutex is a global operating system object! Always close the mutex properly; the best choice is to wrap a mutex object into a `using` block.

This makes it possible to synchronize threads in different programs, which could be useful in a large number of scenarios.

Using the SemaphoreSlim construct

This recipe will show you how to limit multithreaded access to some resources with the help of the `SemaphoreSlim` construct. `SemaphoreSlim` is a lightweight version of `Semaphore`; it limits the number of threads that can access a resource concurrently.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe3`.

How to do it...

To understand how to limit a multithreaded access to a resource with the help of the `SemaphoreSlim` construct, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Below the `Main` method, add the following code snippet:

```
static SemaphoreSlim _semaphore = new SemaphoreSlim(4);

static void AccessDatabase(string name, int seconds)
{
    WriteLine($"{name} waits to access a database");
    _semaphore.Wait();
    WriteLine($"{name} was granted an access to a database");
    Sleep(TimeSpan.FromSeconds(seconds));
    WriteLine($"{name} is completed");
    _semaphore.Release();
}
```

4. Inside the `Main` method, add the following code snippet:

```
for (int i = 1; i <= 6; i++)
{
    string threadName = "Thread " + i;
    int secondsToWait = 2 + 2 * i;
    var t = new Thread(() => AccessDatabase(threadName,
secondsToWait));
    t.Start();
}
```

5. Run the program.

How it works...

When the main program starts, it creates a `SemaphoreSlim` instance, specifying the number of concurrent threads allowed in its constructor. Then, it starts six threads with different names and start times to run.

Every thread tries to acquire access to a database, but we restrict the number of concurrent accesses to a database to four threads with the help of a semaphore. When four threads get access to a database, the other two threads wait until one of the previous threads finishes its work and signals to other threads by calling the `_semaphore.Release` method.

There's more...

Here, we use a hybrid construct, which allows us to save a context switch in cases where the wait time is very short. However, there is an older version of this construct called `Semaphore`. This version is a pure, kernel-time construct. There is no sense in using it, except in one very important scenario; we can create a named semaphore like a named mutex and use it to synchronize threads in different programs. `SemaphoreSlim` does not use Windows kernel semaphores and does not support interprocess synchronization, so use `Semaphore` in this case.

Using the AutoResetEvent construct

In this recipe, there is an example of how to send notifications from one thread to another with the help of an `AutoResetEvent` construct. `AutoResetEvent` notifies a waiting thread that an event has occurred.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe4`.

How to do it...

To understand how to send notifications from one thread to another with the help of the `AutoResetEvent` construct, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;  
using System.Threading;
```

```
using static System.Console;
using static System.Threading.Thread;
```

3. Below the Main method, add the following code snippet:

```
private static AutoResetEvent _workerEvent = new
AutoResetEvent(false);
private static AutoResetEvent _mainEvent = new
AutoResetEvent(false);

static void Process(int seconds)
{
    WriteLine("Starting a long running work...");
    Sleep(TimeSpan.FromSeconds(seconds));
    WriteLine("Work is done!");
    _workerEvent.Set();
    WriteLine("Waiting for a main thread to complete its work");
    _mainEvent.WaitOne();
    WriteLine("Starting second operation...");
    Sleep(TimeSpan.FromSeconds(seconds));
    WriteLine("Work is done!");
    _workerEvent.Set();
}
```

4. Inside the Main method, add the following code snippet:

```
var t = new Thread(() => Process(10));
t.Start();

WriteLine("Waiting for another thread to complete work");
_workerEvent.WaitOne();
WriteLine("First operation is completed!");
WriteLine("Performing an operation on a main thread");
Sleep(TimeSpan.FromSeconds(5));
_mainEvent.Set();
WriteLine("Now running the second operation on a second thread");
_workerEvent.WaitOne();
WriteLine("Second operation is completed!");
```

5. Run the program.

How it works...

When the main program starts, it defines two `AutoResetEvent` instances. One of them is for signaling from the second thread to the main thread, and the second one is for signaling from the main thread to the second thread. We provide `false` to the `AutoResetEvent` constructor, specifying the initial state of both the instances as `unsignaled`. This means that any thread calling the `WaitOne` method of one of these objects will be blocked until we call the `Set` method. If we initialize the event state to `true`, it becomes `signaled` and the first thread calling `WaitOne` will proceed immediately. The event state then becomes `unsignaled` automatically, so we need to call the `Set` method once again to let the other threads calling the `WaitOne` method on this instance to continue.

Then, we create a second thread, which executes the first operation for 10 seconds and waits for the signal from the second thread. The signal notifies that the first operation is completed. Now, the second thread waits for a signal from the main thread. We do some additional work on the main thread and send a signal by calling the `_mainEvent.Set` method. Then, we wait for another signal from the second thread.

`AutoResetEvent` is a kernel-time construct, so if the wait time is not significant, it is better to use the next recipe with `ManualResetEventSlim`, which is a hybrid construct.

Using the ManualResetEventSlim construct

This recipe will describe how to make signaling between threads more flexible with the `ManualResetEventSlim` construct.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe5`.

How to do it...

To understand the use of the `ManualResetEventSlim` construct, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Below the Main method, add the following code:

```
static void TravelThroughGates(string threadName, int seconds)
{
    WriteLine($"{threadName} falls to sleep");
    Sleep(TimeSpan.FromSeconds(seconds));
    WriteLine($"{threadName} waits for the gates to open!");
    _mainEvent.Wait();
    WriteLine($"{threadName} enters the gates!");
}

static ManualResetEventSlim _mainEvent = new
ManualResetEventSlim(false);
```

4. Inside the Main method, add the following code:

```
var t1 = new Thread(() => TravelThroughGates("Thread 1", 5));
var t2 = new Thread(() => TravelThroughGates("Thread 2", 6));
var t3 = new Thread(() => TravelThroughGates("Thread 3", 12));
t1.Start();
t2.Start();
t3.Start();
Sleep(TimeSpan.FromSeconds(6));
WriteLine("The gates are now open!");
_mainEvent.Set();
Sleep(TimeSpan.FromSeconds(2));
_mainEvent.Reset();
WriteLine("The gates have been closed!");
Sleep(TimeSpan.FromSeconds(10));
WriteLine("The gates are now open for the second time!");
_mainEvent.Set();
Sleep(TimeSpan.FromSeconds(2));
WriteLine("The gates have been closed!");
_mainEvent.Reset();
```

5. Run the program.

How it works...

When the main program starts, it first creates an instance of the `ManualResetEventSlim` construct. Then, we start three threads that wait for this event to signal them to continue the execution.

The whole process of working with this construct is like letting people pass through a gate. The `AutoResetEvent` event that we looked at in the previous recipe works like a turnstile, allowing only one person to pass at a time. `ManualResetEventSlim`, which is a hybrid version of `ManualResetEvent`, stays open until we manually call the `Reset` method. Going back to the code, when we call `_mainEvent.Set`, we open it and allow the threads that are ready to accept this signal to continue working. However, thread number three is still sleeping and does not make it in time. We call `_mainEvent.Reset` and we thus close it. The last thread is now ready to go on, but it has to wait for the next signal, which will happen a few seconds later.

There's more...

As in one of the previous recipes, we use a hybrid construct that lacks the possibility to work at the operating system level. If we need to have a global event, we should use the `EventWaitHandle` construct, which is the base class for `AutoResetEvent` and `ManualResetEvent`.

Using the CountdownEvent construct

This recipe will describe how to use the `CountdownEvent` signaling construct to wait until a certain number of operations complete.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe6`.

How to do it...

To understand the use of the `CountdownEvent` construct, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```
3. Below the `Main` method, add the following code:

```
static CountdownEvent _countdown = new CountdownEvent(2);

static void PerformOperation(string message, int seconds)
```

```

{
    Sleep(TimeSpan.FromSeconds(seconds));
    WriteLine(message);
    _countdown.Signal();
}

```

4. Inside the Main method, add the following code:

```

WriteLine("Starting two operations");
var t1 = new Thread(() => PerformOperation("Operation 1 is
completed", 4));
var t2 = new Thread(() => PerformOperation("Operation 2 is
completed", 8));
t1.Start();
t2.Start();
_countdown.Wait();
WriteLine("Both operations have been completed.");
_countdown.Dispose();

```

5. Run the program.

How it works...

When the main program starts, we create a new `CountdownEvent` instance, specifying that we want it to signal when two operations complete in its constructor. Then, we start two threads that signal to the event when they are complete. As soon as the second thread is complete, the main thread returns from waiting on `CountdownEvent` and proceeds further. Using this construct, it is very convenient to wait for multiple asynchronous operations to complete.

However, there is a significant disadvantage; `_countdown.Wait()` will wait forever if we fail to call `_countdown.Signal()` the required number of times. Make sure that all your threads complete with the `Signal` method call when using `CountdownEvent`.

Using the Barrier construct

This recipe illustrates another interesting synchronization construct called `Barrier`. The `Barrier` construct helps to organize several threads so that they meet at some point in time, providing a callback that will be executed each time the threads call the `SignalAndWait` method.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe7`.

How to do it...

To understand the use of the `Barrier` construct, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Below the `Main` method, add the following code:

```
static Barrier _barrier = new Barrier(2,
    b => WriteLine($"End of phase {b.CurrentPhaseNumber + 1}"));

static void PlayMusic(string name, string message, int seconds)
{
    for (int i = 1; i < 3; i++)
    {
        WriteLine("-----");
        Sleep(TimeSpan.FromSeconds(seconds));
        WriteLine($"{name} starts to {message}");
        Sleep(TimeSpan.FromSeconds(seconds));
        WriteLine($"{name} finishes to {message}");
        _barrier.SignalAndWait();
    }
}
```

4. Inside the `Main` method, add the following code:

```
var t1 = new Thread(() => PlayMusic("the guitarist", "play an
    amazing solo", 5));
var t2 = new Thread(() => PlayMusic("the singer", "sing his song",
    2));

t1.Start();
t2.Start();
```

5. Run the program.

How it works...

We create a `Barrier` construct, specifying that we want to synchronize two threads, and after each of those two threads call the `_barrier.SignalAndWait` method, we need to execute a callback that will print out the number of phases completed.

Each thread will send a signal to `Barrier` twice, so we will have two phases. Every time both the threads call the `SignalAndWait` method, `Barrier` will execute the callback. It is useful for working with multithreaded iteration algorithms, to execute some calculations on each iteration end. The end of the iteration is reached when the last thread calls the `SignalAndWait` method.

Using the `ReaderWriterLockSlim` construct

This recipe will describe how to create a thread-safe mechanism to read and write to a collection from multiple threads using a `ReaderWriterLockSlim` construct. `ReaderWriterLockSlim` represents a lock that is used to manage access to a resource, allowing multiple threads for reading or exclusive access for writing.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe8`.

How to do it...

To understand how to create a thread-safe mechanism to read and write to a collection from multiple threads using the `ReaderWriterLockSlim` construct, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Generic;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Below the `Main` method, add the following code:

```
static ReaderWriterLockSlim _rw = new ReaderWriterLockSlim();
static Dictionary<int, int> _items = new Dictionary<int, int>();

static void Read()
{
    WriteLine("Reading contents of a dictionary");
    while (true)
    {
        try
        {
```

```
        _rw.EnterReadLock();
        foreach (var key in _items.Keys)
        {
            Sleep(TimeSpan.FromSeconds(0.1));
        }
    }
    finally
    {
        _rw.ExitReadLock();
    }
}

static void Write(string threadName)
{
    while (true)
    {
        try
        {
            {
                int newKey = new Random().Next(250);
                _rw.EnterUpgradeableReadLock();
                if (!_items.ContainsKey(newKey))
                {
                    try
                    {
                        {
                            _rw.EnterWriteLock();
                            _items[newKey] = 1;
                            WriteLine($"New key {newKey} is added to a dictionary by
a {threadName}");
                        }
                    }
                    finally
                    {
                        {
                            _rw.ExitWriteLock();
                        }
                    }
                }
                Sleep(TimeSpan.FromSeconds(0.1));
            }
        }
        finally
        {
            {
                _rw.ExitUpgradeableReadLock();
            }
        }
    }
}
```

4. Inside the Main method, add the following code:

```
new Thread(Read) { IsBackground = true }.Start();
new Thread(Read) { IsBackground = true }.Start();
new Thread(Read) { IsBackground = true }.Start();

new Thread(() => Write("Thread 1"))
{ IsBackground = true }.Start();
new Thread(() => Write("Thread 2"))
{ IsBackground = true }.Start();

Sleep(TimeSpan.FromSeconds(30));
```

5. Run the program.

How it works...

When the main program starts, it simultaneously runs three threads that read data from a dictionary and two threads that write some data into this dictionary. To achieve thread safety, we use the `ReaderWriterLockSlim` construct, which was designed especially for such scenarios.

It has two kinds of locks: a read lock that allows multiple threads to read and a write lock that blocks every operation from other threads until this write lock is released. There is also an interesting scenario when we obtain a read lock, read some data from the collection, and, depending on that data, decide to obtain a write lock and change the collection. If we get the write locks at once, too much time is spent, not allowing our readers to read the data because the collection is blocked when we get a write lock. To minimize this time, there are `EnterUpgradeableReadLock/ExitUpgradeableReadLock` methods. We get a read lock and read the data; if we find that we have to change the underlying collection, we just upgrade our lock using the `EnterWriteLock` method, then perform a write operation quickly and release a write lock using `ExitWriteLock`.

In our case, we get a random number; we then get a read lock and check whether this number exists in the dictionary key collection. If not, we upgrade our lock to a write lock and then add this new key to a dictionary. It is a good practice to use `try/finally` blocks to make sure that we always release locks after acquiring them.

All our threads have been created as background threads, and after waiting for 30 seconds, the main thread as well as all the background threads get completed.

Using the SpinWait construct

This recipe will describe how to wait on a thread without involving kernel-mode constructs. In addition, we introduce `SpinWait`, a hybrid synchronization construct designed to wait in the user mode for some time, and then switch to the kernel mode to save CPU time.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter2\Recipe9`.

How to do it...

To understand how to wait on a thread without involving kernel-mode constructs, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Below the `Main` method, add the following code:

```
static volatile bool _isCompleted = false;

static void UserModeWait()
{
    while (!_isCompleted)
    {
        Write(".");
    }
    WriteLine();
    WriteLine("Waiting is complete");
}

static void HybridSpinWait()
{
    var w = new SpinWait();
    while (!_isCompleted)
    {
        w.SpinOnce();
    }
}
```

```

        WriteLine(w.NextSpinWillYield);
    }
    WriteLine("Waiting is complete");
}

```

4. Inside the `Main` method, add the following code:

```

var t1 = new Thread(UserModeWait);
var t2 = new Thread(HybridSpinWait);

WriteLine("Running user mode waiting");
t1.Start();
Sleep(20);
_isCompleted = true;
Sleep(TimeSpan.FromSeconds(1));
_isCompleted = false;
WriteLine("Running hybrid SpinWait construct waiting");
t2.Start();
Sleep(5);
_isCompleted = true;

```

5. Run the program.

How it works...

When the main program starts, it defines a thread that will execute an endless loop for 20 milliseconds until the main thread sets the `_isCompleted` variable to `true`. We could experiment and run this cycle for 20-30 seconds instead, measuring the CPU load with the Windows task manager. It will show a significant amount of processor time, depending on how many cores the CPU has.

We use the `volatile` keyword to declare the `_isCompleted` static field. The `volatile` keyword indicates that a field might be modified by multiple threads being executed at the same time. Fields that are declared `volatile` are not subject to compiler and processor optimizations that assume access by a single thread. This ensures that the most up-to-date value is present in the field at all times.

Then, we use a `SpinWait` version, which on each iteration prints a special flag that shows us whether a thread is going to switch to a `blocked` state. We run this thread for 5 milliseconds to see that. In the beginning, `SpinWait` tries to stay in the user mode, and after about nine iterations, it begins to switch the thread to a `blocked` state. If we try to measure the CPU load with this version, we will not see any CPU usage in the Windows task manager.

3

Using a Thread Pool

In this chapter, we will describe the common techniques that are used for working with shared resources from multiple threads. You will learn the following recipes:

- ▶ Invoking a delegate on a thread pool
- ▶ Posting an asynchronous operation on a thread pool
- ▶ A thread pool and the degree of parallelism
- ▶ Implementing a cancellation option
- ▶ Using a wait handle and timeout with a thread pool
- ▶ Using a timer
- ▶ Using the `BackgroundWorker` component

Introduction

In the previous chapters, we discussed several ways to create threads and organize their cooperation. Now, let's consider another scenario where we will create many asynchronous operations that take very little time to complete. As we discussed in the *Introduction* section of *Chapter 1, Threading Basics*, creating a thread is an expensive operation, so doing this for each short-lived, asynchronous operation will include a significant overhead expense.

To deal with this problem, there is a common approach called **pooling** that can be successfully applied to any situation when we need many short-lived, expensive resources. We allocate a certain amount of these resources in advance and organize them into a resource pool. Each time we need a new resource, we just take it from the pool, instead of creating a new one, and return it to the pool after the resource is no longer needed.

The **.NET thread pool** is an implementation of this concept. It is accessible via the `System.Threading.ThreadPool` type. A thread pool is managed by the **.NET Common Language Runtime (CLR)**, which means that there is one instance of a thread pool per CLR. The `ThreadPool` type has a `QueueUserWorkItem` static method that accepts a **delegate**, representing a user-defined, asynchronous operation. After this method is called, this delegate goes to the internal queue. Then, if there are no threads inside the pool, it creates a new **worker thread** and puts the first delegate in the queue on it.

If we put new operations on a thread pool, after the previous operations are completed, it is possible to reuse this one thread to execute these operations. However, if we put new operations faster, the thread pool will create more threads to serve these operations. There is a limit to prevent creating too many threads, and in that case, new operations wait in the queue until the worker threads in the pool become free to serve them.

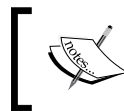


It is very important to keep operations on a thread pool shortlived! Do not put long-running operations on a thread pool or block worker threads. This will lead to all worker threads becoming busy, and they will no longer be able to serve user operations. This, in turn, will lead to performance problems and errors that are very hard to debug.

When we stop putting new operations on a thread pool, it will eventually remove threads that are no longer needed after being idle for some time. This will free up any operating system resources that are no longer required.

I would like to emphasize once again that a thread pool is intended to execute short-running operations. Using a thread pool lets us save operating system resources at the cost of reducing the degree of parallelism. We use fewer threads, but execute asynchronous operations more slowly than usual, batching them by the number of worker threads available. This makes sense if operations complete rapidly, but this will degrade the performance if we execute many long-running, compute-bound operations.

Another important thing to be very careful of is using a thread pool in ASP.NET applications. The ASP.NET infrastructure uses a thread pool itself, and if you waste all worker threads from a thread pool, a web server will no longer be able to serve incoming requests. It is recommended that you use only input/output-bound asynchronous operations in ASP.NET because they use different mechanics called **I/O threads**. We will discuss I/O threads in *Chapter 9, Using Asynchronous I/O*.



Note that worker threads in a thread pool are background threads. This means that when all of the threads in the foreground (including the main application thread) are complete, then all the background threads will be stopped.

In this chapter, you will learn to use a thread pool to execute asynchronous operations. We will cover different ways to put an operation on a thread pool and how to cancel an operation and prevent it from running for a long time.

Invoking a delegate on a thread pool

This recipe will show you how to execute a delegate asynchronously on a thread pool. In addition, we will discuss an approach called the **Asynchronous Programming Model (APM)**, which was historically the first asynchronous programming pattern in .NET.

Getting ready

To step into this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter3\Recipe1`.

How to do it...

To understand how to invoke a delegate on a thread pool, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
private delegate string RunOnThreadPool(out int threadId);

private static void Callback(IAsyncResult ar)
{
    WriteLine("Starting a callback...");
    WriteLine($"State passed to a callbak: {ar.AsyncState}");
    WriteLine($"Is thread pool thread:
{CurrentThread.IsThreadPoolThread}");
    WriteLine($"Thread pool worker thread id:
{CurrentThread.ManagedThreadId}");
}

private static string Test(out int threadId)
{
    WriteLine("Starting...");
    WriteLine($"Is thread pool thread:
{CurrentThread.IsThreadPoolThread}");
    Sleep(TimeSpan.FromSeconds(2));
```

```
        threadId = CurrentThread.ManagedThreadId;
        return $"Thread pool worker thread id was: {threadId}";
    }
}
```

4. Add the following code inside the Main method:

```
int threadId = 0;

RunOnThreadPool poolDelegate = Test;

var t = new Thread(() => Test(out threadId));
t.Start();
t.Join();

WriteLine($"Thread id: {threadId}");

IAsyncResult r = poolDelegate.BeginInvoke(out threadId, Callback,
"a delegate asynchronous call");
r.AsyncWaitHandle.WaitOne();

string result = poolDelegate.EndInvoke(out threadId, r);

WriteLine($"Thread pool worker thread id: {threadId}");
WriteLine(result);

Sleep(TimeSpan.FromSeconds(2));
```

5. Run the program.

How it works...

When the program runs, it creates a thread in the old-fashioned way and then starts it and waits for its completion. Since a thread constructor accepts only a method that does not return any result, we use a **lambda expression** to wrap up a call to the `Test` method. We make sure that this thread is not from the thread pool by printing out the `Thread.CurrentThread.IsThreadPoolThread` property value. We also print out a managed thread ID to identify a thread on which this code was executed.

Then, we define a delegate and run it by calling the `BeginInvoke` method. This method accepts a callback that will be called after the asynchronous operation is complete and a user-defined state to pass into the callback. This state is usually used to distinguish one asynchronous call from another. As a result, we get a `result` object that implements the `IAAsyncResult` interface. The `BeginInvoke` method returns the result immediately, allowing us to continue with any work while the asynchronous operation is being executed on a worker thread of the thread pool. When we need the result of an asynchronous operation, we use the `result` object returned from the `BeginInvoke` method call. We can poll on it using the `IsCompleted` result property, but in this case, we use the `AsyncWaitHandle` result property to wait on it until the operation is complete. After this is done, to get a result from it, we call the `EndInvoke` method on a delegate, passing the delegate arguments and our `IAAsyncResult` object.



Actually, using `AsyncWaitHandle` is not necessary. If we comment out `r.AsyncWaitHandle.WaitOne`, the code will still run successfully because the `EndInvoke` method actually waits for the asynchronous operation to complete. It is always important to call `EndInvoke` (or `EndOperationName` for other asynchronous APIs) because it throws any unhandled exceptions back to the calling thread. Always call both the `Begin` and `End` methods when using this kind of asynchronous API.

When the operation completes, a callback passed to the `BeginInvoke` method will be posted on a thread pool, more specifically, a worker thread. If we comment out the `Thread.Sleep` method call at the end of the `Main` method definition, the callback will not be executed. This is because when the main thread is completed, all the background threads will be stopped, including this callback. It is possible that both asynchronous calls to a delegate and a callback will be served by the same worker thread, which is easy to see by a worker thread ID.

This approach of using the `BeginOperationName/EndOperationName` method and the `IAAsyncResult` object in .NET is called the Asynchronous Programming Model or the APM pattern, and such method pairs are called asynchronous methods. This pattern is still used in various .NET class library APIs, but in modern programming, it is preferable to use the **Task Parallel Library (TPL)** to organize an asynchronous API. We will cover this topic in *Chapter 4, Using the Task Parallel Library*.

Posting an asynchronous operation on a thread pool

This recipe will describe how to put an asynchronous operation on a thread pool.

Getting ready

To step into this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter3\Recipe2`.

How to do it...

To understand how to post an asynchronous operation on a thread pool, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
private static void AsyncOperation(object state)
{
    WriteLine($"Operation state: {state ?? "(null)"}");
    WriteLine($"Worker thread id: {CurrentThread.ManagedThreadId}");
    Sleep(TimeSpan.FromSeconds(2));
}
```

4. Add the following code snippet inside the `Main` method:

```
const int x = 1;
const int y = 2;
const string lambdaState = "lambda state 2";

ThreadPool.QueueUserWorkItem(AsyncOperation);
Sleep(TimeSpan.FromSeconds(1));

ThreadPool.QueueUserWorkItem(AsyncOperation, "async state");
```

```

Sleep(TimeSpan.FromSeconds(1));

ThreadPool.QueueUserWorkItem( state =>
{
    WriteLine($"Operation state: {state}");
    WriteLine($"Worker thread id: {CurrentThread.ManagedThreadId}");
    Sleep(TimeSpan.FromSeconds(2));
}, "lambda state");

ThreadPool.QueueUserWorkItem( _ =>
{
    WriteLine($"Operation state: {x + y}, {lambdaState}");
    WriteLine($"Worker thread id: {CurrentThread.ManagedThreadId}");
    Sleep(TimeSpan.FromSeconds(2));
}, "lambda state");

Sleep(TimeSpan.FromSeconds(2));

```

5. Run the program.

How it works...

First, we define the `AsyncOperation` method that accepts a single parameter of the object type. Then, we post this method on a thread pool using the `QueueUserWorkItem` method. Then, we post this method once again, but this time, we pass a `state` object to this method call. This object will be passed to the `AsynchronousOperation` method as the `state` parameter.

Making a thread sleep for 1 second after these operations allows the thread pool to reuse threads for new operations. If you comment on these `Thread.Sleep` calls, most certainly the thread IDs will be different in all cases. If not, probably the first two threads will be reused to run the following two operations.

First, we post a lambda expression to a thread pool. Nothing special here; instead of defining a separate method, we use the lambda expression syntax.

Secondly, instead of passing the state of a lambda expression, we use **closure** mechanics. This gives us more flexibility and allows us to provide more than one object to the asynchronous operation and static typing for those objects. So, the previous mechanism of passing an object into a method callback is really redundant and obsolete. There is no need to use it now when we have closures in C#.

A thread pool and the degree of parallelism

This recipe will show you how a thread pool works with many asynchronous operations and how it is different from creating many separate threads.

Getting ready

To step into this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found in `BookSamples\Chapter3\Recipe3`.

How to do it...

To learn how a thread pool works with many asynchronous operations and how it is different from creating many separate threads, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Diagnostics;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void UseThreads(int numberOfOperations)
{
    using (var countdown = new CountdownEvent(numberOfOperations))
    {
        WriteLine("Scheduling work by creating threads");
        for (int i = 0; i < numberOfOperations; i++)
        {
            var thread = new Thread(() =>
            {
                Write($"{CurrentThread.ManagedThreadId}, ");
                Sleep(TimeSpan.FromSeconds(0.1));
                countdown.Signal();
            });
            thread.Start();
        }
        countdown.Wait();
    }
}
```

```

        WriteLine();
    }
}

static void UseThreadPool(int numberOfOperations)
{
    using (var countdown = new CountdownEvent(numberOfOperations))
    {
        WriteLine("Starting work on a threadpool");
        for (int i = 0; i < numberOfOperations; i++)
        {
            ThreadPool.QueueUserWorkItem( _ =>
            {
                Write($"{CurrentThread.ManagedThreadId},");
                Sleep(TimeSpan.FromSeconds(0.1));
                countdown.Signal();
            });
        }
        countdown.Wait();
        WriteLine();
    }
}

```

4. Add the following code snippet inside the Main method:

```

const int numberOfOperations = 500;
var sw = new Stopwatch();
sw.Start();
UseThreads(numberOfOperations);
sw.Stop();
WriteLine($"Execution time using threads:
{sw.ElapsedMilliseconds}");

sw.Reset();
sw.Start();
UseThreadPool(numberOfOperations);
sw.Stop();
WriteLine($"Execution time using the thread pool:
{sw.ElapsedMilliseconds}");

```

5. Run the program.

How it works...

When the main program starts, we create many different threads and run an operation on each one of them. This operation prints out a thread ID and blocks a thread for 100 milliseconds. As a result, we create 500 threads running all these operations in parallel. The total time on my machine is about 300 milliseconds, but we consume many operating system resources for all these threads.

Then, we follow the same workflow, but instead of creating a thread for each operation, we post them on a thread pool. After this, the thread pool starts to serve these operations; it begins to create more threads near the end; however, it still takes much more time, about 12 seconds on my machine. We save memory and threads for operating system use but pay for it with application performance.

Implementing a cancellation option

This recipe shows an example on how to cancel an asynchronous operation on a thread pool.

Getting ready

To step into this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found in `BookSamples\Chapter3\Recipe4`.

How to do it...

To understand how to implement a cancellation option on a thread, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void AsyncOperation1(CancellationToken token)
{
    WriteLine("Starting the first task");
    for (int i = 0; i < 5; i++)
    {
        if (token.IsCancellationRequested)
```

```
        {
            WriteLine("The first task has been canceled.");
            return;
        }
        Sleep(TimeSpan.FromSeconds(1));
    }
    WriteLine("The first task has completed succesfully");
}

static void AsyncOperation2(Cancellation_token token)
{
    try
    {
        WriteLine("Starting the second task");

        for (int i = 0; i < 5; i++)
        {
            token.ThrowIfCancellationRequested();
            Sleep(TimeSpan.FromSeconds(1));
        }
        WriteLine("The second task has completed succesfully");
    }
    catch (OperationCanceledException)
    {
        WriteLine("The second task has been canceled.");
    }
}

static void AsyncOperation3(Cancellation_token token)
{
    bool cancellationFlag = false;
    token.Register(() => cancellationFlag = true);
    WriteLine("Starting the third task");
    for (int i = 0; i < 5; i++)
    {
        if (cancellationFlag)
        {
            WriteLine("The third task has been canceled.");
            return;
        }
        Sleep(TimeSpan.FromSeconds(1));
    }
    WriteLine("The third task has completed succesfully");
}
```

4. Add the following code snippet inside the Main method:

```
using (var cts = new CancellationTokenSource())
{
    CancellationToken token = cts.Token;
    ThreadPool.QueueUserWorkItem(_ => AsyncOperation1(token));
    Sleep(TimeSpan.FromSeconds(2));
    cts.Cancel();
}

using (var cts = new CancellationTokenSource())
{
    CancellationToken token = cts.Token;
    ThreadPool.QueueUserWorkItem(_ => AsyncOperation2(token));
    Sleep(TimeSpan.FromSeconds(2));
    cts.Cancel();
}

using (var cts = new CancellationTokenSource())
{
    CancellationToken token = cts.Token;
    ThreadPool.QueueUserWorkItem(_ => AsyncOperation3(token));
    Sleep(TimeSpan.FromSeconds(2));
    cts.Cancel();
}

Sleep(TimeSpan.FromSeconds(2));
```

5. Run the program.

How it works...

Here, we introduce the `CancellationTokenSource` and `CancellationToken` constructs. They appeared in .NET 4.0 and now are the de facto standard to implement asynchronous operation cancellation processes. Since the thread pool has existed for a long time, it has no special API for cancellation tokens; however, they can still be used.

In this program, we see three ways to organize a cancellation process. The first is just to poll and check the `CancellationToken.IsCancellationRequested` property. If it is set to `true`, this means that our operation is being cancelled and we must abandon the operation.

The second way is to throw an `OperationCanceledException` exception. This allows us to control the cancellation process not from inside the operation, which is being canceled, but from the code on the outside.

The last option is to register a **callback** that will be called on a thread pool when an operation is canceled. This will allow us to chain cancellation logic into another asynchronous operation.

Using a wait handle and timeout with a thread pool

This recipe will describe how to implement a timeout for thread pool operations and how to wait properly on a thread pool.

Getting ready

To step into this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter3\Recipe5`.

How to do it...

To learn how to implement a timeout and how to wait properly on a thread pool, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void RunOperations(TimeSpan workerOperationTimeout)
{
    using (var evt = new ManualResetEvent(false))
    using (var cts = new CancellationTokenSource())
    {
        WriteLine("Registering timeout operation...");
        var worker = ThreadPool.RegisterWaitForSingleObject(evt
            , (state, isTimedOut) => WorkerOperationWait(cts,
isTimedOut)
            , null
            , workerOperationTimeout
            , true);

        WriteLine("Starting long running operation...");
        ThreadPool.QueueUserWorkItem(_ => WorkerOperation(cts.Token,
evt));

        Sleep(workerOperationTimeout.Add(TimeSpan.FromSeconds(2)));
    }
}
```

```
        worker.Unregister(evt);
    }
}

static void WorkerOperation(Cancellation token,
ManualResetEvent evt)
{
    for(int i = 0; i < 6; i++)
    {
        if (token.IsCancellationRequested)
        {
            return;
        }
        Sleep(TimeSpan.FromSeconds(1));
    }
    evt.Set();
}

static void WorkerOperationWait(Cancellation tokenSource cts, bool
isTimedOut)
{
    if (isTimedOut)
    {
        cts.Cancel();
        WriteLine("Worker operation timed out and was canceled.");
    }
    else
    {
        WriteLine("Worker operation succeeded.");
    }
}
```

4. Add the following code snippet inside the Main method:

```
RunOperations(TimeSpan.FromSeconds(5));
RunOperations(TimeSpan.FromSeconds(7));
```

5. Run the program.

How it works...

A thread pool has another useful method: `ThreadPool.RegisterWaitForSingleObject`. This method allows us to queue a callback on a thread pool, and this callback will be executed when the provided wait handle is signaled or a timeout has occurred. This allows us to implement a timeout for thread pool operations.

First, we register the timeout handling asynchronous operation. It will be called when one of the following events take place: on receiving a signal from the `ManualResetEvent` object, which is set by the worker operation when it is completed successfully, or when a timeout has occurred before the first operation is completed. If this happens, we use `CancellationToken` to cancel the first operation.

Then, we queue a long-running worker operation on a thread pool. It runs for 6 seconds and then sets a `ManualResetEvent` signaling construct, in case it completes successfully. In other case, if the cancellation is requested, the operation is just abandoned.

Finally, if we provide a 5-second timeout for the operation, that would not be enough. This is because the operation takes 6 seconds to complete, and we'd need to cancel this operation. So, if we provide a 7-second timeout, which is acceptable, the operation completes successfully.

There's more...

This is very useful when you have a large number of threads that must wait in the `blocked` state for some multithreaded event construct to signal. Instead of blocking all these threads, we are able to use the thread pool infrastructure. It will allow us to free up these threads until the event is set. This is a very important scenario for server applications, which require scalability and performance.

Using a timer

This recipe will describe how to use a `System.Threading.Timer` object to create periodically-called asynchronous operations on a thread pool.

Getting ready

To step into this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter3\Recipe6`.

How to do it...

To learn how to create periodically-called asynchronous operations on a thread pool, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static Timer _timer;

static void TimerOperation(DateTime start)
{
    TimeSpan elapsed = DateTime.Now - start;
    WriteLine($"{elapsed.Seconds} seconds from {start}. " +
        $"Timer thread pool thread id: {CurrentThread.ManagedThreadId}");
}
```

4. Add the following code snippet inside the `Main` method:

```
WriteLine("Press 'Enter' to stop the timer...");
DateTime start = DateTime.Now;
_timer = new Timer(_ => TimerOperation(start), null
    , TimeSpan.FromSeconds(1)
    , TimeSpan.FromSeconds(2));
try
{
    Sleep(TimeSpan.FromSeconds(6));

    _timer.Change(TimeSpan.FromSeconds(1), TimeSpan.FromSeconds(4));

    ReadLine();
}
finally
{
    _timer.Dispose();
}
```

5. Run the program.

How it works...

First, we create a new `Timer` instance. The first parameter is a lambda expression that will be executed on a thread pool. We call the `TimerOperation` method, providing it with a start date. We do not use the user state object, so the second parameter is null; then, we specify when are we going to run `TimerOperation` for the first time and what will be the period between calls. So, the first value actually means that we start the first operation in 1 second, and then, we run each of them in 2 seconds.

After this, we wait for 6 seconds and change our timer. We start `TimerOperation` in a second after calling the `_timer.Change` method, and then run each of them for 4 seconds.



A timer could be more complex than this!

It is possible to use a timer in more complicated ways. For instance, we can run the timer operation only once, by providing a timer period parameter with the `Timeout.Infinite` value. Then, inside the timer asynchronous operation, we are able to set the next time when the timer operation will be executed, depending on some custom logic.

Lastly, we wait for the *Enter* key to be pressed and to finish the application. While it is running, we can see the time passed since the program started.

Using the BackgroundWorker component

This recipe describes another approach to asynchronous programming via an example of a `BackgroundWorker` component. With the help of this object, we are able to organize our asynchronous code as a set of events and event handlers. You will learn how to use this component for asynchronous programming.

Getting ready

To step into this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter3\Recipe7`.

How to do it...

To learn how to use the `BackgroundWorker` component, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.ComponentModel;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void Worker_DoWork(object sender, DoWorkEventArgs e)
{
    WriteLine($"DoWork thread pool thread id:
{CurrentThread.ManagedThreadId}");
    var bw = (BackgroundWorker) sender;
    for (int i = 1; i <= 100; i++)
```



```
{
    if (bw.CancellationPending)
    {
        e.Cancel = true;
        return;
    }
    if (i%10 == 0)
    {
        bw.ReportProgress(i);
    }

    Sleep(TimeSpan.FromSeconds(0.1));
}

e.Result = 42;
}

static void Worker_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
    WriteLine($"{e.ProgressPercentage}% completed. " +
        $"Progress thread pool thread id:
{CurrentThread.ManagedThreadId}");
}

static void Worker_Completed(object sender,
RunWorkerCompletedEventArgs e)
{
    WriteLine($"Completed thread pool thread id:
{CurrentThread.ManagedThreadId}");
    if (e.Error != null)
    {
        WriteLine($"Exception {e.Error.Message} has occurred.");
    }
    else if (e.Cancelled)
    {
        WriteLine($"Operation has been canceled.");
    }
    else
    {
        WriteLine($"The answer is: {e.Result}");
    }
}
```

4. Add the following code snippet inside the `Main` method:

```
var bw = new BackgroundWorker();
bw.WorkerReportsProgress = true;
bw.WorkerSupportsCancellation = true;

bw.DoWork += Worker_DoWork;
bw.ProgressChanged += Worker_ProgressChanged;
bw.RunWorkerCompleted += Worker_Completed;

bw.RunWorkerAsync();

WriteLine("Press C to cancel work");
do
{
    if (ReadKey(true).KeyChar == 'C')
    {
        bw.CancelAsync();
    }
}
while (bw.IsBusy);
```

5. Run the program.

How it works...

When the program starts, we create an instance of a `BackgroundWorker` component. We explicitly state that we want our background worker to support cancellation and notifications on the operation's progress.

Now, this is where the most interesting part comes into play. Instead of manipulating with a thread pool and delegates, we use another C# idiom called **events**. An event represents a *source* of notifications and a number of *subscribers* ready to react when a notification arrives. In our case, we state that we will subscribe for three events, and when they occur, we call the corresponding **event handlers**. These are methods with a specially defined signature that will be called when an event notifies its subscribers.

Therefore, instead of organizing an asynchronous API in a pair of `Begin/End` methods, it is possible to just start an asynchronous operation and then subscribe to different events that could happen while this operation is executed. This approach is called an **Event-based Asynchronous Pattern (EAP)**. It was historically the second attempt to structure asynchronous programs, and now, it is recommended to use TPL instead, which will be described in *Chapter 4, Using the Task Parallel Library*.

So, we subscribed to three events. The first of them is the `DoWork` event. A handler of this event will be called when a background worker object starts an asynchronous operation with the `RunWorkerAsync` method. The event handler will be executed on a thread pool, and this is the main operating point where work is canceled if cancellation is requested and where we provide information on the progress of the operation. At last, when we get the result, we set it to event arguments, and then, the `RunWorkerCompleted` event handler is called. Inside this method, we find out whether our operation has succeeded, there were some errors, or it was canceled.

Besides this, a `BackgroundWorker` component is actually intended to be used in **Windows Forms Applications (WPF)**. Its implementation makes working with UI controls possible from a background worker's event handler code directly, which is very comfortable as compared to the interaction of worker threads in a thread pool with UI controls.

4

Using the Task Parallel Library

In this chapter, we will dive into a new asynchronous programming paradigm, the Task Parallel Library. You will learn the following recipes:

- ▶ Creating a task
- ▶ Performing basic operations with a task
- ▶ Combining tasks together
- ▶ Converting the APM pattern to tasks
- ▶ Converting the EAP pattern to tasks
- ▶ Implementing a cancelation option
- ▶ Handling exceptions in tasks
- ▶ Running tasks in parallel
- ▶ Tweaking the execution of tasks with `TaskScheduler`

Introduction

In the previous chapters, you learned what a thread is, how to use threads, and why we need a thread pool. Using a thread pool allows us to save operating system resources at the cost of reducing a parallelism degree. We can think of a thread pool as an **abstraction layer** that hides details of thread usage from a programmer, allowing us to concentrate on a program's logic rather than on threading issues.

However, using a thread pool is complicated as well. There is no easy way to get a result from a thread pool worker thread. We need to implement our own way to get a result back, and in case of an exception, we have to propagate it to the original thread properly. Besides this, there is no easy way to create a set of dependent asynchronous actions, where one action runs after another finishes its work.

There were several attempts to work around these issues, which resulted in the creation of the Asynchronous Programming Model and the Event-based Asynchronous Pattern, mentioned in *Chapter 3, Using a Thread Pool*. These patterns made getting results easier and did a good job of propagating exceptions, but combining asynchronous actions together still required a lot of work and resulted in a large amount of code.

To resolve all these problems, a new API for asynchronous operations was introduced in .Net Framework 4.0. It was called the Task Parallel Library (TPL). It was changed slightly in .Net Framework 4.5 and to make it clear, we will work with the latest version of TPL using the 4.6 version of .Net Framework in our projects. TPL can be considered as one more abstraction layer over a thread pool, hiding the lower-level code that will work with the thread pool from a programmer and supplying a more convenient and fine-grained API.

The core concept of TPL is a task. A task represents an asynchronous operation that can be run in a variety of ways, using a separate thread or not. We will look through all the possibilities in detail in this chapter.



By default, a programmer is not aware of how exactly a task is being executed. TPL raises the level of abstraction by hiding the task implementation details from the user. Unfortunately, in some cases, this could lead to mysterious errors, such as the application hanging while trying to get a result from the task. This chapter will help you understand the mechanics under the hood of TPL and how to avoid using it in improper ways.

A task can be combined with other tasks in different variations. For example, we are able to start several tasks simultaneously, wait for all of them to complete, and then run a task that will perform some calculations over all the previous tasks' results. Convenient APIs for task combination are one of the key advantages of TPL compared to the previous patterns.

There are also several ways to deal with exceptions resulting from tasks. Since a task may consist of several other tasks, and they in turn have their child tasks as well, there is the concept of `AggregateException`. This type of exception holds all exceptions from underlying tasks inside it, allowing us to handle them separately.

And, last but not least, C# has built-in support for TPL since version 5.0, allowing us to work with tasks in a very smooth and comfortable way using the new `await` and `async` keywords. We will discuss this topic in *Chapter 5, Using C# 6.0*.

In this chapter, you will learn to use TPL to execute asynchronous operations. We will learn what a task is, cover different ways to create tasks, and will learn how to combine tasks. We will also discuss how to convert legacy APM and EAP patterns to use tasks, how to handle exceptions properly, how to cancel tasks, and how to work with several tasks that are being executed simultaneously. In addition, we will find out how to deal with tasks in Windows GUI applications properly.

Creating a task

This recipe shows the basic concept of what a task is. You will learn how to create and execute tasks.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe1`.

How to do it...

To create and execute a task, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.



This time, make sure that you are using .Net Framework 4.5 or higher for every project.

2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void TaskMethod(string name)
{
    WriteLine($"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. Is thread pool thread:
" +
        $"{CurrentThread.IsThreadPoolThread}");
}
```

4. Add the following code snippet inside the `Main` method:

```
var t1 = new Task(() => TaskMethod("Task 1"));
var t2 = new Task(() => TaskMethod("Task 2"));
t2.Start();
t1.Start();
Task.Run(() => TaskMethod("Task 3"));
Task.Factory.StartNew(() => TaskMethod("Task 4"));
Task.Factory.StartNew(() => TaskMethod("Task 5"),
TaskCreationOptions.LongRunning);
Sleep(TimeSpan.FromSeconds(1));
```

5. Run the program.

How it works...

When the program runs, it creates two tasks with its constructor. We pass the lambda expression as the `Action` delegate; this allows us to provide a string parameter to `TaskMethod`. Then, we run these tasks using the `Start` method.



Note that until we call the `Start` method on these tasks, they will not start execution. It is very easy to forget to actually start the task.

Then, we run two more tasks using the `Task.Run` and `Task.Factory.StartNew` methods. The difference is that both the created tasks immediately start working, so we do not need to call the `Start` method on the tasks explicitly. All of the tasks, numbered `Task 1` to `Task 4`, are placed on thread pool worker threads and run in an unspecified order. If you run the program several times, you will find that the task execution order is not defined.

The `Task.Run` method is just a shortcut to `Task.Factory.StartNew`, but the latter method has additional options. In general, use the former method unless you need to do something special, as in the case of `Task 5`. We mark this task as long-running, and as a result, this task will be run on a separate thread that does not use a thread pool. However, this behavior could change, depending on the current **task scheduler** that runs the task. You will learn what a task scheduler is in the last recipe of this chapter.

Performing basic operations with a task

This recipe will describe how to get the result value from a task. We will go through several scenarios to understand the difference between running a task on a thread pool or on a main thread.

Getting ready

To start this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe2`.

How to do it...

To perform basic operations with a task, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static Task<int> CreateTask(string name)
{
    return new Task<int>(() => TaskMethod(name));
}

static int TaskMethod(string name)
{
    WriteLine($"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}");
    Sleep(TimeSpan.FromSeconds(2));
    return 42;
}
```

4. Add the following code snippet inside the `Main` method:

```
TaskMethod("Main Thread Task");
Task<int> task = CreateTask("Task 1");
task.Start();
int result = task.Result;
WriteLine($"Result is: {result}");

task = CreateTask("Task 2");
task.RunSynchronously();
result = task.Result;
WriteLine($"Result is: {result}");
```



```
task = CreateTask("Task 3");
WriteLine(task.Status);
task.Start();

while (!task.IsCompleted)
{
    WriteLine(task.Status);
    Sleep(TimeSpan.FromSeconds(0.5));
}

WriteLine(task.Status);
result = task.Result;
WriteLine($"Result is: {result}");
```

5. Run the program.

How it works...

At first, we run `TaskMethod` without wrapping it into a task. As a result, it is executed synchronously, providing us with information about the main thread. Obviously, it is not a thread pool thread.

Then, we run `Task 1`, starting it with the `Start` method and waiting for the result. This task will be placed on a thread pool, and the main thread waits and is blocked until the task returns.

We do the same with `Task 2`, except that we run it using the `RunSynchronously()` method. This task will run on the main thread, and we get exactly the same output as in the very first case when we called `TaskMethod` synchronously. This is a very useful optimization that allows us to avoid thread pool usage for very short-lived operations.

We run `Task 3` in the same way we did `Task 1`, but instead of blocking the main thread, we just spin, printing out the task status until the task is completed. This shows several task statuses, which are `Created`, `Running`, and `RanToCompletion`, respectively.

Combining tasks

This recipe will show you how to set up tasks that are dependent on each other. We will learn how to create a task that will run after the parent task is complete. In addition, we will discover a way to save thread usage for very short-lived tasks.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe3`.

How to do it...

To combine tasks, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static int TaskMethod(string name, int seconds)
{
    WriteLine(
        $"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}");
    Sleep(TimeSpan.FromSeconds(seconds));
    return 42 * seconds;
}
```

4. Add the following code snippet inside the `Main` method:

```
var firstTask = new Task<int>(() => TaskMethod("First Task", 3));
var secondTask = new Task<int>(() => TaskMethod("Second Task",
2));

firstTask.ContinueWith(
    t => WriteLine(
        $"The first answer is {t.Result}. Thread id " +
        $"{CurrentThread.ManagedThreadId}, is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}",
        TaskContinuationOptions.OnlyOnRanToCompletion);

firstTask.Start();
secondTask.Start();

Sleep(TimeSpan.FromSeconds(4));

Task continuation = secondTask.ContinueWith(
    t => WriteLine(
        $"The second answer is {t.Result}. Thread id " +
```

```
        $"{CurrentThread.ManagedThreadId}, is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}"),
        TaskContinuationOptions.OnlyOnRanToCompletion
        | TaskContinuationOptions.ExecuteSynchronously);

continuation.GetAwaiter().OnCompleted(
    () => WriteLine(
        $"Continuation Task Completed! Thread id " +
        $"{CurrentThread.ManagedThreadId}, is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}"));

Sleep(TimeSpan.FromSeconds(2));
WriteLine();

firstTask = new Task<int>(() =>
{
    var innerTask = Task.Factory.StartNew(() => TaskMethod("Second
Task", 5), TaskCreationOptions.AttachedToParent);

    innerTask.ContinueWith(t => TaskMethod("Third Task", 2),
        TaskContinuationOptions.AttachedToParent);

    return TaskMethod("First Task", 2);
});

firstTask.Start();

while (!firstTask.IsCompleted)
{
    WriteLine(firstTask.Status);
    Sleep(TimeSpan.FromSeconds(0.5));
}
WriteLine(firstTask.Status);

Sleep(TimeSpan.FromSeconds(10));
```

5. Run the program.

How it works...

When the main program starts, we create two tasks, and for the first task, we set up a **continuation** (a block of code that runs after the antecedent task is complete). Then, we start both tasks and wait for 4 seconds, which is enough for both tasks to be complete. Then, we run another continuation to the second task and try to execute it synchronously by specifying a `TaskContinuationOptions.ExecuteSynchronously` option. This is a useful technique when the continuation is very short lived, and it will be faster to run it on the main thread than to put it on a thread pool. We are able to achieve this because the second task is completed by that moment. If we comment out the 4-second `Thread.Sleep` method, we will see that this code will be put on a thread pool because we do not have the result from the antecedent task yet.

Finally, we define a continuation for the previous continuation, but in a slightly different manner, using the new `GetAwaiter` and `OnCompleted` methods. These methods are intended to be used along with C# language asynchronous mechanics. We will cover this topic later in *Chapter 5, Using C# 6.0*.

The last part of the demo is about the parent-child task relationships. We create a new task, and while running this task, we run a so-called child task by providing a `TaskCreationOptions.AttachedToParent` option.



The child task must be created while running a parent task so that it is attached to the parent properly!

This means that the parent task will not be complete until all child tasks finish their work. We are also able to run continuations on those child tasks that provide a `TaskContinuationOptions.AttachedToParent` option. These continuation tasks will affect the parent task as well, and it will not be complete until the very last child task ends.

Converting the APM pattern to tasks

In this recipe, we will see how to convert an old-fashioned APM API to a task. There are examples of different situations that could take place in the process of conversion.

Getting ready

To start this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe4`.

How to do it...

To convert the APM pattern to tasks, carry out the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
delegate string AsynchronousTask(string threadName);
delegate string IncompatibleAsynchronousTask(out int threadId);

static void Callback(IAsyncResult ar)
{
    WriteLine("Starting a callback...");
    WriteLine($"State passed to a callbak: {ar.AsyncState}");
    WriteLine($"Is thread pool thread:
{CurrentThread.IsThreadPoolThread}");
    WriteLine($"Thread pool worker thread id:
{CurrentThread.ManagedThreadId}");
}

static string Test(string threadName)
{
    WriteLine("Starting...");
    WriteLine($"Is thread pool thread:
{CurrentThread.IsThreadPoolThread}");
    Sleep(TimeSpan.FromSeconds(2));
    CurrentThread.Name = threadName;
    return $"Thread name: {CurrentThread.Name}";
}

static string Test(out int threadId)
{
    WriteLine("Starting...");
    WriteLine($"Is thread pool thread:
{CurrentThread.IsThreadPoolThread}");
    Sleep(TimeSpan.FromSeconds(2));
    threadId = CurrentThread.ManagedThreadId;
    return $"Thread pool worker thread id was: {threadId}";
}
```

4. Add the following code snippet inside the Main method:

```
int threadId;
AsyncTask d = Test;
IncompatibleAsyncTask e = Test;

WriteLine("Option 1");
Task<string> task = Task<string>.Factory.FromAsync(
    d.BeginInvoke("AsyncTaskThread", Callback,
        "a delegate asynchronous call"), d.EndInvoke);

task.ContinueWith(t => WriteLine(
    $"Callback is finished, now running a continuation! Result:
    {t.Result}"));

while (!task.IsCompleted)
{
    WriteLine(task.Status);
    Sleep(TimeSpan.FromSeconds(0.5));
}
WriteLine(task.Status);
Sleep(TimeSpan.FromSeconds(1));

WriteLine("-----");
WriteLine();
WriteLine("Option 2");

task = Task<string>.Factory.FromAsync(
    d.BeginInvoke, d.EndInvoke, "AsyncTaskThread",
    "a delegate asynchronous call");

task.ContinueWith(t => WriteLine(
    $"Task is completed, now running a continuation! Result:
    {t.Result}"));
while (!task.IsCompleted)
{
    WriteLine(task.Status);
    Sleep(TimeSpan.FromSeconds(0.5));
}
WriteLine(task.Status);
Sleep(TimeSpan.FromSeconds(1));

WriteLine("-----");
WriteLine();
```

```
WriteLine("Option 3");

IAsyncResult ar = e.BeginInvoke(out threadId, Callback,
    "a delegate asynchronous call");
task = Task<string>.Factory.FromAsync(ar, _ =>
    e.EndInvoke(out threadId, ar));

task.ContinueWith(t =>
    WriteLine(
        $"Task is completed, now running a continuation! " +
        $"Result: {t.Result}, ThreadId: {threadId}"));

while (!task.IsCompleted)
{
    WriteLine(task.Status);
    Sleep(TimeSpan.FromSeconds(0.5));
}
WriteLine(task.Status);

Sleep(TimeSpan.FromSeconds(1));
```

5. Run the program.

How it works...

Here, we define two kinds of delegates; one of them uses the `out` parameter and therefore is incompatible with the standard TPL API for converting the APM pattern to tasks. Then, we have three examples of such a conversion.

The key point for converting APM to TPL is the `Task<T>.Factory.FromAsync` method, where `T` is the asynchronous operation result type. There are several overloads of this method; in the first case, we pass `IAsyncResult` and `Func<IAsyncResult, string>`, which is a method that accepts the `IAsyncResult` implementation and returns a string. Since the first delegate type provides `EndMethod`, which is compatible with this signature, we have no problem converting this delegate asynchronous call to a task.

In the second example, we do almost the same, but use a different `FromAsync` method overload, which does not allow specifying a callback that will be executed after the asynchronous delegate call is completed. We are able to replace this with a continuation, but if the callback is important, we can use the first example.

The last example shows a little trick. This time, `EndMethod` of the `IncompatibleAsynchronousTask` delegate uses the `out` parameter and is not compatible with any `FromAsync` method overload. However, it is very easy to wrap the `EndMethod` call into a lambda expression that will be suitable for the task factory.

To see what is going on with the underlying task, we are printing its status while waiting for the asynchronous operation's result. We see that the first task's status is `WaitingForActivation`, which means that the task has not actually been started yet by the TPL infrastructure.

Converting the EAP pattern to tasks

This recipe will describe how to translate event-based asynchronous operations to tasks. In this recipe, you will find a solid pattern that is suitable for every event-based asynchronous API in the .NET Framework class library.

Getting ready

To begin this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe5`.

How to do it...

To convert the EAP pattern to tasks, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.ComponentModel;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static int TaskMethod(string name, int seconds)
{
    WriteLine(
        $"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}");

    Sleep(TimeSpan.FromSeconds(seconds));
    return 42 * seconds;
}
```


4. Add the following code snippet inside the Main method:

```
var tcs = new TaskCompletionSource<int>();

var worker = new BackgroundWorker();
worker.DoWork += (sender, EventArgs) =>
{
    EventArgs.Result = TaskMethod("Background worker", 5);
};

worker.RunWorkerCompleted += (sender, EventArgs) =>
{
    if (EventArgs.Error != null)
    {
        tcs.SetException(EventArgs.Error);
    }
    else if (EventArgs.Cancelled)
    {
        tcs.SetCanceled();
    }
    else
    {
        tcs.SetResult((int)EventArgs.Result);
    }
};

worker.RunWorkerAsync();

int result = tcs.Task.Result;

WriteLine($"Result is: {result}");
```

5. Run the program.

How it works...

This is a very simple and elegant example of converting EAP patterns to tasks. The key point is to use the `TaskCompletionSource<T>` type, where `T` is an asynchronous operation result type.

It is also important not to forget to wrap the `tcs.SetResult` method call into the `try/catch` block in order to guarantee that the error information is always set to the task completion source object. It is also possible to use the `TrySetResult` method instead of `SetResult` to make sure that the result has been set successfully.

Implementing a cancelation option

This recipe is about implementing the cancelation process for task-based asynchronous operations. You will learn how to use the cancelation token properly for tasks and how to find out whether a task is canceled before it was actually run.

Getting ready

To start with this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe6`.

How to do it...

To implement a cancelation option for task-based asynchronous operations, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static int TaskMethod(string name, int seconds,
CancellationToken token)
{
    WriteLine(
        $"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}");

    for (int i = 0; i < seconds; i++)
    {
        Sleep(TimeSpan.FromSeconds(1));
        if (token.IsCancellationRequested) return -1;
    }
    return 42*seconds;
}
```

4. Add the following code snippet inside the `Main` method:

```
var cts = new CancellationTokenSource();
var longTask = new Task<int>(() =>
    TaskMethod("Task 1", 10, cts.Token), cts.Token);
WriteLine(longTask.Status);
cts.Cancel();
WriteLine(longTask.Status);
WriteLine("First task has been cancelled before execution");

cts = new CancellationTokenSource();
longTask = new Task<int>(() =>
    TaskMethod("Task 2", 10, cts.Token), cts.Token);
longTask.Start();
for (int i = 0; i < 5; i++ )
{
    Sleep(TimeSpan.FromSeconds(0.5));
    WriteLine(longTask.Status);
}
cts.Cancel();
for (int i = 0; i < 5; i++)
{
    Sleep(TimeSpan.FromSeconds(0.5));
    WriteLine(longTask.Status);
}

WriteLine($"A task has been completed with result {longTask.
Result}.");
```

5. Run the program.

How it works...

This is another very simple example of how to implement the cancelation option for a TPL task. You are already familiar with the cancelation token concept we discussed in *Chapter 3, Using a Thread Pool*.

First, let's look closely at the `longTask` creation code. We're providing a cancelation token to the underlying task once and then to the task constructor for the second time. *Why do we need to supply this token twice?*

The answer is that if we cancel the task before it was actually started, its TPL infrastructure is responsible for dealing with the cancelation because our code will not be executed at all. We know that the first task was canceled by getting its status. If we try to call the `Start` method on this task, we will get `InvalidOperationException`.

Then, we deal with the cancelation process from our own code. This means that we are now fully responsible for the cancelation process, and after we canceled the task, its status was still `RanToCompletion` because from TPL's perspective, the task finished its job normally. It is very important to distinguish these two situations and understand the responsibility difference in each case.

Handling exceptions in tasks

This recipe describes the very important topic of handling exceptions in asynchronous tasks. We will go through different aspects of what happens to exceptions thrown from tasks and how to get to their information.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe7`.

How to do it...

To handle exceptions in tasks, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static int TaskMethod(string name, int seconds)
{
    WriteLine(
        $"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}");

    Sleep(TimeSpan.FromSeconds(seconds));
    throw new Exception("Boom!");
    return 42 * seconds;
}
```

4. Add the following code snippet inside the Main method:

```
Task<int> task;
try
{
    task = Task.Run(() => TaskMethod("Task 1", 2));
    int result = task.Result;
    WriteLine($"Result: {result}");
}
catch (Exception ex)
{
    WriteLine($"Exception caught: {ex}");
}
WriteLine("-----");
WriteLine();

try
{
    task = Task.Run(() => TaskMethod("Task 2", 2));
    int result = task.GetAwaiter().GetResult();
    WriteLine($"Result: {result}");
}
catch (Exception ex)
{
    WriteLine($"Exception caught: {ex}");
}
WriteLine("-----");
WriteLine();

var t1 = new Task<int>(() => TaskMethod("Task 3", 3));
var t2 = new Task<int>(() => TaskMethod("Task 4", 2));
var complexTask = Task.WhenAll(t1, t2);
var exceptionHandler = complexTask.ContinueWith(t =>
    WriteLine($"Exception caught: {t.Exception}"),
    TaskContinuationOptions.OnlyOnFaulted
);
t1.Start();
t2.Start();

Sleep(TimeSpan.FromSeconds(5));
```

5. Run the program.

How it works...

When the program starts, we create a task and try to get the task results synchronously. The `Get` part of the `Result` property makes the current thread wait until the completion of the task and propagates the exception to the current thread. In this case, we easily catch the exception in a catch block, but this exception is a wrapper exception called `AggregateException`. In this case, it holds only one exception inside because only one task has thrown this exception, and it is possible to get the underlying exception by accessing the `InnerException` property.

The second example is mostly the same, but to access the task result, we use the `GetAwaiter` and `GetResult` methods. In this case, we do not have a wrapper exception because it is unwrapped by the TPL infrastructure. We have an original exception at once, which is quite comfortable if we have only one underlying task.

The last example shows the situation where we have two task-throwing exceptions. To handle exceptions, we now use a continuation, which is executed only in case the antecedent task finishes with an exception. This behavior is achieved by providing a `TaskContinuationOptions.OnlyOnFaulted` option to a continuation. As a result, we have `AggregateException` being printed out, and we have two inner exceptions from both the tasks inside it.

There's more...

As tasks may be connected in a very different manner, the resulting `AggregateException` exception might contain other aggregate exceptions inside along with the usual exceptions. Those inner aggregate exceptions might themselves contain other aggregate exceptions within them.

To get rid of those wrappers, we should use the root aggregate exception's `Flatten` method. It will return a collection of all the inner exceptions of every child aggregate exception in the hierarchy.

Running tasks in parallel

This recipe shows how to handle many asynchronous tasks that are running simultaneously. You will learn how to be notified effectively when all tasks are complete or any of the running tasks have to finish their work.

Getting ready

To start this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe8`.

How to do it...

To run tasks in parallel, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static int TaskMethod(string name, int seconds)
{
    WriteLine(
        $"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. Is thread pool thread: " +
        $"{CurrentThread.IsThreadPoolThread}");

    Sleep(TimeSpan.FromSeconds(seconds));
    return 42 * seconds;
}
```

4. Add the following code snippet inside the `Main` method:

```
var firstTask = new Task<int>(() => TaskMethod("First Task", 3));
var secondTask = new Task<int>(() =>
    TaskMethod("Second Task", 2));
var whenAllTask = Task.WhenAll(firstTask, secondTask);

whenAllTask.ContinueWith(t =>
    WriteLine($"The first answer is {t.Result[0]}, the second is {t.Result[1]}"),
    TaskContinuationOptions.OnlyOnRanToCompletion);

firstTask.Start();
secondTask.Start();

Sleep(TimeSpan.FromSeconds(4));

var tasks = new List<Task<int>>();
for (int i = 1; i < 4; i++)
{
    int counter = i;
```

```

        var task = new Task<int>(() =>
        TaskMethod($"Task {counter}", counter));
        tasks.Add(task);
        task.Start();
    }

    while (tasks.Count > 0)
    {
        var completedTask = Task.WhenAny(tasks).Result;
        tasks.Remove(completedTask);
        WriteLine
        ($"A task has been completed with result {completedTask.Result}.");
    }

    Sleep(TimeSpan.FromSeconds(1));

```

5. Run the program.

How it works...

When the program starts, we create two tasks, and then, with the help of the `Task.WhenAll` method, we create a third task, which will be complete after all initial tasks are complete. The resulting task provides us with an answer array, where the first element holds the first task's result, the second element holds the second result, and so on.

Then, we create another list of tasks and wait for any of those tasks to be completed with the `Task.WhenAny` method. After we have one finished task, we remove it from the list and continue to wait for the other tasks to be complete until the list is empty. This method is useful to get the task completion progress or to use a timeout while running the tasks. For example, we wait for a number of tasks, and one of those tasks is counting a timeout. If this task is completed first, we just cancel all other tasks that are not completed yet.

Tweaking the execution of tasks with TaskScheduler

This recipe describes another very important aspect of dealing with tasks, which is a proper way to work with a UI from the asynchronous code. You will learn what a task scheduler is, why it is so important, how it can harm our application, and how to use it to avoid errors.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter4\Recipe9`.

How to do it...

To tweak task execution with `TaskScheduler`, perform the following steps:

1. Start Visual Studio 2015. Create a new C# **WPF Application** project. This time, we will need a UI thread with a message loop, which is not available in console applications.
2. In the `MainWindow.xaml` file, add the following markup inside a grid element (that is, between the `<Grid>` and `</Grid>` tags):

```
<TextBlock Name="ContentTextBlock"
HorizontalAlignment="Left"
Margin="44,134,0,0"
VerticalAlignment="Top"
Width="425"
Height="40"/>
<Button Content="Sync"
HorizontalAlignment="Left"
Margin="45,190,0,0"
VerticalAlignment="Top"
Width="75"
Click="ButtonSync_Click"/>
<Button Content="Async"
HorizontalAlignment="Left"
Margin="165,190,0,0"
VerticalAlignment="Top"
Width="75"
Click="ButtonAsync_Click"/>
<Button Content="Async OK"
HorizontalAlignment="Left"
Margin="285,190,0,0"
VerticalAlignment="Top"
Width="75"
Click="ButtonAsyncOK_Click"/>
```

3. In the `MainWindow.xaml.cs` file, use the following using directives:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
```

4. Add the following code snippet below the `MainWindow` constructor:

```
void ButtonSync_Click(object sender, RoutedEventArgs e)
{
    ContentTextBlock.Text = string.Empty;
```

```
try
{
    //string result = TaskMethod(
    // TaskScheduler.FromCurrentSynchronizationContext()).Result;
    string result = TaskMethod().Result;
    ContentTextBlock.Text = result;
}
catch (Exception ex)
{
    ContentTextBlock.Text = ex.InnerException.Message;
}
}

void ButtonAsync_Click(object sender, RoutedEventArgs e)
{
    ContentTextBlock.Text = string.Empty;
    Mouse.OverrideCursor = Cursors.Wait;
    Task<string> task = TaskMethod();
    task.ContinueWith(t =>
    {
        ContentTextBlock.Text = t.Exception.InnerException.Message;
        Mouse.OverrideCursor = null;
    },
    CancellationToken.None,
    TaskContinuationOptions.OnlyOnFaulted,
    TaskScheduler.FromCurrentSynchronizationContext());
}

void ButtonAsyncOK_Click(object sender, RoutedEventArgs e)
{
    ContentTextBlock.Text = string.Empty;
    Mouse.OverrideCursor = Cursors.Wait;
    Task<string> task = TaskMethod(
        TaskScheduler.FromCurrentSynchronizationContext());

    task.ContinueWith(t => Mouse.OverrideCursor = null,
        CancellationToken.None,
        TaskContinuationOptions.None,
        TaskScheduler.FromCurrentSynchronizationContext());
}

Task<string> TaskMethod()
{
    return TaskMethod(TaskScheduler.Default);
}
```

```
}

Task<string> TaskMethod(TaskScheduler scheduler)
{
    Task delay = Task.Delay(TimeSpan.FromSeconds(5));

    return delay.ContinueWith(t =>
    {
        string str =
            "Task is running on a thread id " +
            $"{CurrentThread.ManagedThreadId}. Is thread pool thread:
" +
            $"{CurrentThread.IsThreadPoolThread}";

        ContentTextBlock.Text = str;
        return str;
    }, scheduler);
}
```

5. Run the program.

How it works...

Here, we meet many new things. First, we created a WPF application instead of a console application. It is necessary because we need a user interface thread with a message loop to demonstrate the different options of running a task asynchronously.

There is a very important abstraction called `TaskScheduler`. This component is actually responsible for how the task will be executed. The default task scheduler puts tasks on a thread pool worker thread. This is the most common scenario; unsurprisingly, it is the default option in TPL. We also know how to run a task synchronously and how to attach them to the parent tasks to run those tasks together. Now, let's see what else we can do with tasks.

When the program starts, we create a window with three buttons. The first button invokes a synchronous task execution. The code is placed inside the `ButtonSync_Click` method. While the task runs, we are not even able to move the application window. The user interface gets totally frozen while the user interface thread is busy running the task and cannot respond to any message loop until the task is complete. This is quite a common bad practice for GUI Windows applications, and we need to find a way to work around this issue.

The second problem is that we try to access the UI controls from another thread. Graphical User Interface controls have never been designed to be used from multiple threads, and to avoid possible errors, you are not allowed to access these components from a thread other than the one on which it was created. When we try to do that, we get an exception, and the exception message is printed on the main window in 5 seconds.

To resolve the first problem, we try to run the task asynchronously. This is what the second button does; the code for this is placed inside the `ButtonAsync_Click` method. If you run the task in a debugger, you will see that it is placed on a thread pool, and in the end, we will get the same exception. However, the user interface remains responsive all the time the task runs. This is a good thing, but we need to get rid of the exception.

And we already did that! To output the error message, a continuation was provided with the `TaskScheduler.FromCurrentSynchronizationContext` option. If this wasn't done, we would not see the error message because we would get the same exception that took place inside the task. This option instructs the TPL infrastructure to put a code inside the continuation on the UI thread and run it asynchronously with the help of the UI thread message loop. This resolves the problem with accessing UI controls from another thread, but still keeps our UI responsive.

To check whether this is true, we press the last button that runs the code inside the `ButtonAsyncOK_Click` method. All that is different is that we provide the UI thread task scheduler to our task. After the task is complete, you will see that it runs on the UI thread in an asynchronous manner. The UI remains responsive, and it is even possible to press another button despite the wait cursor being active.

However, there are some tricks to use the UI thread in order to run tasks. If we go back to the synchronous task code and uncomment the line with getting the result with the UI thread task scheduler provided, we will never get any result. This is a classical deadlock situation: we are dispatching an operation in the queue of the UI thread, and the UI thread waits for this operation to complete, but as it waits, it cannot run the operation, which will never end (or even start). This will also happen if we call the `Wait` method on a task. To avoid deadlock, never use synchronous operations on a task scheduled to the UI thread; just use `ContinueWith` or `async/await` from C#.

5

Using C# 6.0

In this chapter, we will look through native asynchronous programming support in the C# 6.0 programming language. You will learn the following recipes:

- ▶ Using the `await` operator to get asynchronous task results
- ▶ Using the `await` operator in a lambda expression
- ▶ Using the `await` operator with consequent asynchronous tasks
- ▶ Using the `await` operator for the execution of parallel asynchronous tasks
- ▶ Handling exceptions in asynchronous operations
- ▶ Avoiding the use of the captured synchronization context
- ▶ Working around the `async void` method
- ▶ Designing a custom awaitable type
- ▶ Using the `dynamic` type with `await`

Introduction

Until now, you learned about the Task Parallel Library, the latest asynchronous programming infrastructure from Microsoft. It allows us to design our program in a modular manner, combining different asynchronous operations together.

Unfortunately, it is still difficult to understand the actual program flow when reading such a program. In a large program, there will be numerous tasks and continuations that depend on each other, continuations that run other continuations, and continuations for exception handling. They are all gathered together in the program code in very different places. Therefore, understanding the sequence of which operation goes first and what happens next becomes a very challenging problem.

Another issue to watch out for is whether the proper synchronization context is propagated to each asynchronous task that could touch user interface controls. It is only permitted to use these controls from the UI thread; otherwise, we would get a multithreaded access exception.

Speaking about exceptions, we also have to use separate continuation tasks to handle errors that occur inside antecedent asynchronous operation or operations. This in turn results in complicated error-handling code that is spread through different parts of the code, not logically related to each other.

To address these issues, the authors of C# introduced new language enhancements called **asynchronous functions** along with C# version 5.0. They really make asynchronous programming simple, but at the same time, it is a higher level abstraction over TPL. As we mentioned in *Chapter 4, Using the Task Parallel Library*, abstraction hides important implementation details and makes asynchronous programming easier at the cost of taking away many important things from a programmer. It is very important to understand the concept behind asynchronous functions to create robust and scalable applications.


To create an asynchronous function, you first mark a method with the `async` keyword. It is not possible to have the `async` property or event accessor methods and constructors without doing this first. The code will look as follows:

```
async Task<string> GetStringAsync()
{
    await Task.Delay(TimeSpan.FromSeconds(2));
    return "Hello, World!";
}
```

Another important fact is that asynchronous functions must return the `Task` or `Task<T>` type. It is possible to have `async void` methods, but it is preferable to use the `async Task` method instead. The only reasonable option to use `async void` functions is when using top-level UI control event handlers in your application.

Inside a method marked with the `async` keyword, you can use the `await` operator. This operator works with tasks from TPL and gets the result of the asynchronous operation inside the task. The details will be covered later in the chapter. You cannot use the `await` operator outside the `async` method; there will be a compilation error. In addition, asynchronous functions should have at least one `await` operator inside their code. However, not having an `await` operator will lead to just a compilation warning, not an error.

It is important to note that this method returns immediately after the line with the `await` call. In case of a synchronous execution, the executing thread will be blocked for 2 seconds and then return a result. Here, we wait asynchronously while returning a worker thread to a thread pool immediately after executing the `await` operator. After 2 seconds, we get the worker thread from a thread pool once again and run the rest of the asynchronous method on it. This allows us to reuse this worker thread to do some other work while these 2 seconds pass, which is extremely important for application scalability. With the help of asynchronous functions, we have a linear program control flow, but it is still asynchronous. This is both very comfortable and very confusing. The recipes in this chapter will help you learn every important aspect of asynchronous functions.

 In my experience, there is a common misunderstanding about how programs work if there are two consecutive `await` operators in it. Many people think that if we use the `await` function on one asynchronous operation after another, they run in parallel. However, they actually run sequentially; the second one starts only when the first operation completes. It is very important to remember this, and later in the chapter, we will cover this topic in detail.

There are a number of limitations connected with using `async` and `await` operators. In C# 5.0, for example, it is not possible to mark the console application's `Main` method as `async`; you cannot have the `await` operator inside a `catch`, `finally`, `lock`, or `unsafe` block. It is not allowed to have `ref` and `out` parameters on an asynchronous function. There are more subtleties, but these are the major points. In C# 6.0, some of these limitations have been removed; you can use `await` inside `catch` and `finally` blocks due to compiler internal enhancements.

Asynchronous functions are turned into complex program constructs by the C# compiler behind the scenes. I intentionally will not describe this in detail; the resulting code is quite similar to another C# construct, called **iterators**, and is implemented as a sort of state machine. Since many developers have started using the `async` modifier almost on every method, I would like to emphasize that there is no sense in marking a method `async` if it is not intended to be used in an asynchronous or parallel manner. Calling the `async` method includes a significant performance hit, and the usual method call is going to be about 40 to 50 times faster as compared to the same method marked with the `async` keyword. Please be aware of that.

In this chapter, you will learn to use the C# `async` and `await` keywords to work with asynchronous operations. We will cover how to await asynchronous operations sequentially and parallelly. We will discuss how to use `await` in lambda expressions, how to handle exceptions, and how to avoid pitfalls when using the `async void` methods. To conclude the chapter, we will dive deep into synchronization context propagation and you will learn how to create your own awaitable objects instead of using tasks.

Using the await operator to get asynchronous task results

This recipe walks you through the basic scenario of using asynchronous functions. We will compare how to get an asynchronous operation result with TPL and with the `await` operator.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe1`.

How to do it...

To use the `await` operator in order to get asynchronous task results, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static Task AsynchronyWithTPL()
{
    Task<string> t = GetInfoAsync("Task 1");
    Task t2 = t.ContinueWith(task => WriteLine(t.Result),
        TaskContinuationOptions.NotOnFaulted);
    Task t3 = t.ContinueWith(task =>
        WriteLine(t.Exception.InnerException),
        TaskContinuationOptions.OnlyOnFaulted);

    return Task.WhenAny(t2, t3);
}

static async Task AsynchronyWithAwait()
{
    try
    {
        string result = await GetInfoAsync("Task 2");
        WriteLine(result);
    }
}
```

```

    }
    catch (Exception ex)
    {
        WriteLine(ex);
    }
}

static async Task<string> GetInfoAsync(string name)
{
    await Task.Delay(TimeSpan.FromSeconds(2));
    //throw new Exception("Boom!");
    return
        $"Task {name} is running on a thread id {CurrentThread.
ManagedThreadId}." +
        $" Is thread pool thread: {CurrentThread.IsThreadPoolThread}";
}

```

4. Add the following code snippet inside the `Main` method:

```

Task t = AsynchronyWithTPL();
t.Wait();

t = AsynchronyWithAwait();
t.Wait();

```

5. Run the program.

How it works...

When the program runs, we run two asynchronous operations. One of them is standard TPL-powered code and the second one uses the new `async` and `await` C# features. The `AsynchronyWithTPL` method starts a task that runs for 2 seconds and then returns a string with information about the worker thread. Then, we define a continuation to print out the asynchronous operation result after the operation is complete and another one to print the exception details in case errors occur. Finally, we return a task representing one of the continuation tasks and wait for its completion in the `Main` method.

In the `AsynchronyWithAwait` method, we achieve the same result by using `await` with the task. It is as if we write just the usual synchronous code—we get the result from the task, print out the result, and catch an exception if the task is completed with errors. The key difference is that we actually have an asynchronous program. Immediately after using `await`, C# creates a task that has a continuation task with all the remaining code after the `await` operator and deals with exception propagation as well. Then, we return this task to the `Main` method and wait until it gets completed.



Note that depending on the nature of the underlying asynchronous operation and the current synchronization context, the exact means of executing asynchronous code may differ. We will explain this later in the chapter.

Therefore, we can see that the first and the second parts of the program are conceptually equivalent, but in the second part the C# compiler does the work of handling asynchronous code implicitly. It is, in fact, even more complicated than the first part, and we will cover the details in the next few recipes of this chapter.

Remember that it is not recommended to use the `Task.Wait` and `Task.Result` methods in environments such as the Windows GUI or ASP.NET. This could lead to deadlocks if the programmer is not 100% aware of what is really going on in the code. This was illustrated in the *Tweaking the execution of tasks with TaskScheduler* recipe in *Chapter 4, Using the Task Parallel Library*, when we used `Task.Result` in the WPF application.

To test how exception handling works, just uncomment the `throw new Exception` line inside the `GetInfoAsync` method.

Using the await operator in a lambda expression

This recipe will show you how to use `await` inside a lambda expression. We will write an anonymous method that uses `await` and get a result of the method execution asynchronously.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe2`.

How to do it...

To write an anonymous method that uses `await` and get a result of the method execution asynchronously using the `await` operator in a lambda expression, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static async Task AsynchronousProcessing()
{
    Func<string, Task<string>> asyncLambda = async name => {
        await Task.Delay(TimeSpan.FromSeconds(2));
        return
            $"Task {name} is running on a thread id {CurrentThread.
ManagedThreadId}." +
            $" Is thread pool thread: {CurrentThread.IsThreadPoolThread}";
    };

    string result = await asyncLambda("async lambda");

    WriteLine(result);
}
```

4. Add the following code snippet inside the `Main` method:

```
Task t = AsynchronousProcessing();
t.Wait();
```

5. Run the program.

How it works...

First, we move out the asynchronous function into the `AsynchronousProcessing` method, since we cannot use `async` with `Main`. Then, we describe a lambda expression using the `async` keyword. As the type of any lambda expression cannot be inferred from lambda itself, we have to specify its type to the C# compiler explicitly. In our case, the type means that our lambda expression accepts one string parameter and returns a `Task<string>` object.

Then, we define the lambda expression body. One aberration is that the method is defined to return a `Task<string>` object, but we actually return a string and get no compilation errors! The C# compiler automatically generates a task and returns it for us.

The last step is to await the asynchronous lambda expression execution and print out the result.

Using the await operator with consequent asynchronous tasks

This recipe will show you how exactly the program flows when we have several consecutive `await` methods in the code. You will learn how to read the code with the `await` method and understand why the `await` call is an asynchronous operation.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe3`.

How to do it...

To understand a program flow in the presence of consecutive `await` methods, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static Task AsynchronyWithTPL()
{
    var containerTask = new Task(() => {
        Task<string> t = GetInfoAsync("TPL 1");
        t.ContinueWith(task => {
            WriteLine(t.Result);
            Task<string> t2 = GetInfoAsync("TPL 2");
            t2.ContinueWith(innerTask => WriteLine(innerTask.Result),
                TaskContinuationOptions.NotOnFaulted |
                TaskContinuationOptions.AttachedToParent);
            t2.ContinueWith(innerTask =>
                WriteLine(innerTask.Exception.InnerException),
                TaskContinuationOptions.OnlyOnFaulted |
                TaskContinuationOptions.AttachedToParent);
        },
        TaskContinuationOptions.NotOnFaulted |
        TaskContinuationOptions.AttachedToParent);
    });
}
```

```

        t.ContinueWith(task => WriteLine(t.Exception.InnerException),
            TaskContinuationOptions.OnlyOnFaulted |
            TaskContinuationOptions.AttachedToParent);
    });

    containerTask.Start();
    return containerTask;
}

static async Task AsynchronyWithAwait()
{
    try
    {
        string result = await GetInfoAsync("Async 1");
        WriteLine(result);
        result = await GetInfoAsync("Async 2");
        WriteLine(result);
    }
    catch (Exception ex)
    {
        WriteLine(ex);
    }
}

static async Task<string> GetInfoAsync(string name)
{
    WriteLine($"Task {name} started!");
    await Task.Delay(TimeSpan.FromSeconds(2));
    if (name == "TPL 2")
        throw new Exception("Boom!");
    return
        $"Task {name} is running on a thread id {CurrentThread.
ManagedThreadId}." +
        $" Is thread pool thread: {CurrentThread.IsThreadPoolThread}";
}

```

4. Add the following code snippet inside the Main method:

```

Task t = AsynchronyWithTPL();
t.Wait();

t = AsynchronyWithAwait();
t.Wait();

```

5. Run the program.

How it works...

When the program runs, we run two asynchronous operations just as we did in the first recipe. However, this time, we shall start from the `AsynchronyWithAwait` method. It still looks like the usual synchronous code; the only difference is the two `await` statements. The most important point is that the code is still sequential, and the `Async 2` task will start only after the previous one is completed. When we read the code, the program flow is very clear: we see what runs first and what goes after. Then, how is this program asynchronous? Well, first, it is not always asynchronous. If a task is already complete when we use `await`, we will get its result synchronously. Otherwise, the common approach when we see an `await` statement inside the code is to note that at this point, the method will return immediately and the rest of the code will be run in a continuation task. Since we do not block the execution, waiting for the result of an operation, it is an asynchronous call. Instead of calling `t.Wait` in the `Main` method, we can perform any other task while the code in the `AsynchronyWithAwait` method is being executed. However, the main thread must wait until all the asynchronous operations complete, or they will be stopped as they run on background threads.

The `AsynchronyWithTPL` method imitates the same program flow as the `AsynchronyWithAwait` method does. We need a container task to handle all the dependent tasks together. Then, we start the main task and add a set of continuations to it. When the task is complete, we print out the result; we then start one more task, which in turn has more continuations to continue work after the second task is complete. To test the exception handling, we throw an exception on purpose when running the second task and get its information printed out. This set of continuations creates the same program flow as in the first method, and when we compare it to the code with the `await` methods, we can see that it is much easier to read and understand. The only trick is to remember that asynchrony does not always mean parallel execution.

Using the await operator for the execution of parallel asynchronous tasks

In this recipe, you will learn how to use `await` to run asynchronous operations in parallel instead of the usual sequential execution.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe4`.

How to do it...

To understand the use of the `await` operator for parallel asynchronous task execution, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code below the `Main` method:

```
static async Task AsynchronousProcessing()
{
    Task<string> t1 = GetInfoAsync("Task 1", 3);
    Task<string> t2 = GetInfoAsync("Task 2", 5);

    string[] results = await Task.WhenAll(t1, t2);
    foreach (string result in results)
    {
        WriteLine(result);
    }
}

static async Task<string> GetInfoAsync(string name, int seconds)
{
    await Task.Delay(TimeSpan.FromSeconds(seconds));
    //await Task.Run(() =>
    //    Thread.Sleep(TimeSpan.FromSeconds(seconds)));
    return
        $"Task {name} is running on a thread id " +
        $"{CurrentThread.ManagedThreadId}. " +
        $"Is thread pool thread: {CurrentThread.IsThreadPoolThread}";
}
```

4. Add the following code snippet inside the `Main` method:

```
Task t = AsynchronousProcessing();
t.Wait();
```

5. Run the program.

How it works...

Here, we define two asynchronous tasks running for 3 and 5 seconds, respectively. Then, we use a `Task.WhenAll` helper method to create another task that will be complete only when all of the underlying tasks get completed. Then, we await the result of this combined task. After 5 seconds, we get all the results, which means that the tasks were running simultaneously.

However, there is one interesting observation. When you run the program, you might note that both tasks are likely to be served by the same worker thread from a thread pool. How is this possible when we have run the tasks in parallel? To make things even more interesting, let's comment out the `await Task.Delay` line inside the `GetIntroAsync` method and uncomment the `await Task.Run` line, and then run the program.

We will see that in this case, both the tasks will be served by different worker threads. The difference is that `Task.Delay` uses a timer under the hood, and the processing goes as follows: we get the worker thread from a thread pool, which awaits the `Task.Delay` method to return a result. Then, the `Task.Delay` method starts the timer and specifies a piece of code that will be called when the timer counts the number of seconds specified to the `Task.Delay` method. Then, we immediately return the worker thread to a thread pool. When the timer event runs, we get any available worker thread from a thread pool once again (which could be the same thread that we used first) and run the code provided to the timer on it.

When we use the `Task.Run` method, we get a worker thread from a thread pool and make it block for a number of seconds, provided to the `Thread.Sleep` method. Then, we get a second worker thread and block it as well. In this scenario, we consume two worker threads and they do absolutely nothing, as they are not able to perform any other task while waiting.

We will talk in detail about the first scenario in *Chapter 9, Using Asynchronous I/O*, where we will discuss a large set of asynchronous operations working with data inputs and outputs. Using the first approach whenever possible is the key to creating scalable server applications.

Handling exceptions in asynchronous operations

This recipe will describe how to deal with exception handling using asynchronous functions in C#. You will learn how to work with aggregate exceptions in case you use `await` with multiple parallel asynchronous operations.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe5`.

How to do it...

To understand handling exceptions in asynchronous operations, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
```

3. Add the following code snippet below the `Main` method:

```
static async Task AsynchronousProcessing()
{
    WriteLine("1. Single exception");

    try
    {
        string result = await GetInfoAsync("Task 1", 2);
        WriteLine(result);
    }
    catch (Exception ex)
    {
        WriteLine($"Exception details: {ex}");
    }

    WriteLine();
    WriteLine("2. Multiple exceptions");

    Task<string> t1 = GetInfoAsync("Task 1", 3);
    Task<string> t2 = GetInfoAsync("Task 2", 2);
    try
    {
        string[] results = await Task.WhenAll(t1, t2);
        WriteLine(results.Length);
    }
    catch (Exception ex)
    {
        WriteLine($"Exception details: {ex}");
    }

    WriteLine();
    WriteLine("3. Multiple exceptions with AggregateException");
}
```

```
t1 = GetInfoAsync("Task 1", 3);
t2 = GetInfoAsync("Task 2", 2);
Task<string[]> t3 = Task.WhenAll(t1, t2);
try
{
    string[] results = await t3;
    WriteLine(results.Length);
}
catch
{
    var ae = t3.Exception.Flatten();
    var exceptions = ae.InnerExceptions;
    WriteLine($"Exceptions caught: {exceptions.Count}");
    foreach (var e in exceptions)
    {
        WriteLine($"Exception details: {e}");
        WriteLine();
    }
}

WriteLine();
    WriteLine("4. await in catch and finally blocks");

try
{
    string result = await GetInfoAsync("Task 1", 2);
    WriteLine(result);
}
catch (Exception ex)
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    WriteLine($"Catch block with await: Exception details: {ex}");
}
finally
{
    await Task.Delay(TimeSpan.FromSeconds(1));
    WriteLine("Finally block");
}
}

static async Task<string> GetInfoAsync(string name, int seconds)
{
    await Task.Delay(TimeSpan.FromSeconds(seconds));
    throw new Exception($"Boom from {name}!");
}
```

4. Add the following code snippet inside the `Main` method:

```
Task t = AsynchronousProcessing();  
t.Wait();
```

5. Run the program.

How it works...

We run four scenarios to illustrate the most common cases of error handling using `async` and `await` in C#. The first case is very simple and almost identical to the usual synchronous code. We just use the `try/catch` statement and get the exception's details.

A very common mistake is using the same approach when more than one asynchronous operations are being awaited. If we use the `catch` block in the same way as we did before, we will get only the first exception from the underlying `AggregateException` object.

To collect all the information, we have to use the awaited tasks' `Exception` property. In the third scenario, we flatten the `AggregateException` hierarchy and then unwrap all the underlying exceptions from it using the `Flatten` method of `AggregateException`.

To illustrate C# 6.0 changes, we use `await` inside `catch` and `finally` blocks of the exception handling code. To verify that it was not possible to use `await` inside `catch` and `finally` blocks in the previous version of C#, you can compile it against C# 5.0 by specifying it in the project properties under the build section advanced settings.

Avoiding the use of the captured synchronization context

This recipe discusses the details of the synchronization context behavior when `await` is used to get asynchronous operation results. You will learn how and when to turn off the synchronization context flow.

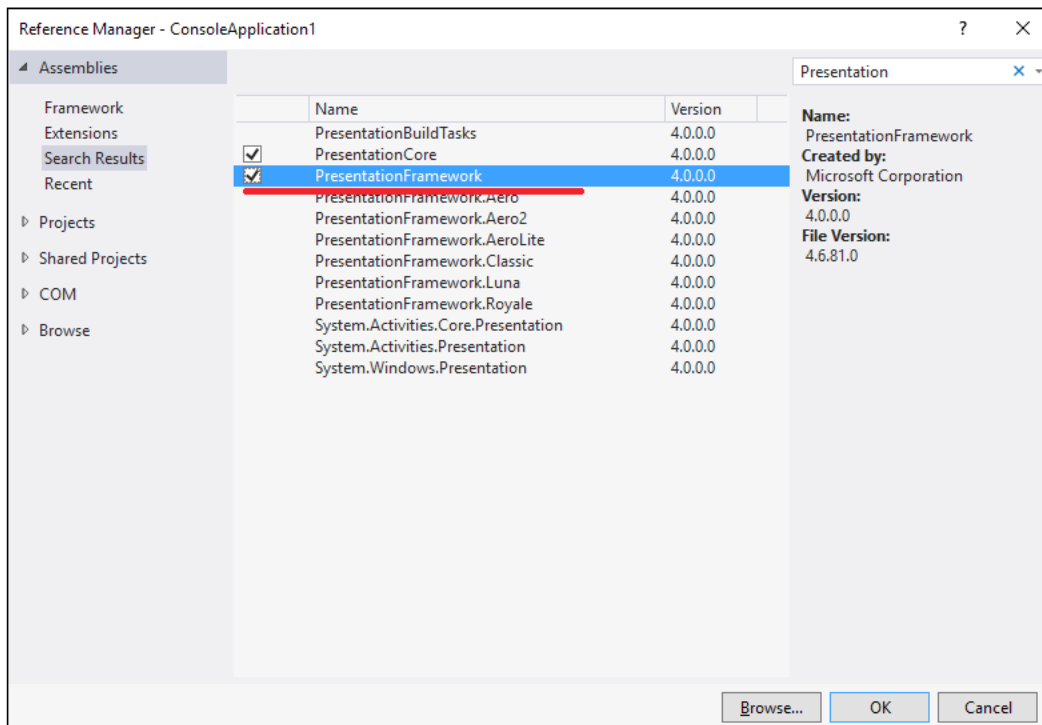
Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe6`.

How to do it...

To understand the details of the synchronization context behavior when `await` is used and to learn how and when to turn off the synchronization context flow, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add references to the Windows Presentation Foundation Library by following these steps:
 1. Right-click on the **References** folder in the project, and select the **Add reference...** menu option.
 2. Add references to these libraries: **PresentationCore**, **PresentationFramework**, **System.Xaml**, and **WindowsBase**. You can use the search function in the reference manager dialog as follows:



3. In the `Program.cs` file, add the following `using` directives:

```
using System;  
using System.Diagnostics;  
using System.Text;  
using System.Threading.Tasks;
```

```
using System.Windows;
using System.Windows.Controls;
using static System.Console;
```

4. Add the following code snippet below the Main method:

```
private static Label _label;

static async void Click(object sender, EventArgs e)
{
    _label.Content = new TextBlock {Text = "Calculating..."};
    TimeSpan resultWithContext = await Test();
    TimeSpan resultNoContext = await TestNoContext();
    //TimeSpan resultNoContext =
    //    await TestNoContext().ConfigureAwait(false);
    var sb = new StringBuilder();
    sb.AppendLine($"With the context: {resultWithContext}");
    sb.AppendLine($"Without the context: {resultNoContext}");
    sb.AppendLine("Ratio: " +
        $"{resultWithContext.TotalMilliseconds/resultNoContext.
TotalMilliseconds:0.00}");
    _label.Content = new TextBlock {Text = sb.ToString()};
}

static async Task<TimeSpan> Test()
{
    const int iterationsNumber = 100000;
    var sw = new Stopwatch();
    sw.Start();
    for (int i = 0; i < iterationsNumber; i++)
    {
        var t = Task.Run(() => { });
        await t;
    }
    sw.Stop();
    return sw.Elapsed;
}

static async Task<TimeSpan> TestNoContext()
{
    const int iterationsNumber = 100000;
    var sw = new Stopwatch();
    sw.Start();
    for (int i = 0; i < iterationsNumber; i++)
    {
```

```
var t = Task.Run(() => { });
await t.ConfigureAwait(
    continueOnCapturedContext: false);
}
sw.Stop();
return sw.Elapsed;
}
```

5. Replace the Main method with the following code snippet:

```
[STAThread]
static void Main(string[] args)
{
    var app = new Application();
    var win = new Window();
    var panel = new StackPanel();
    var button = new Button();
    _label = new Label();
    _label.FontSize = 32;
    _label.Height = 200;
    button.Height = 100;
    button.FontSize = 32;
    button.Content = new TextBlock {
        Text = "Start asynchronous operations" };
    button.Click += Click;
    panel.Children.Add(_label);
    panel.Children.Add(button);
    win.Content = panel;
    app.Run(win);

    ReadLine();
}
```

6. Run the program.

How it works...

In this example, we studied one of the most important aspects of an asynchronous function's default behavior. You already know about task schedulers and synchronization contexts from *Chapter 4, Using the Task Parallel Library*. By default, the `await` operator tries to capture synchronization contexts and executes the preceding code on it. As we already know, this helps us write asynchronous code by working with user interface controls. In addition, deadlock situations, such as those that were described in the previous chapter, will not happen when using `await`, since we do not block the UI thread while waiting for the result.

This is reasonable, but let's see what can potentially happen. In this example, we create a Windows Presentation Foundation application programmatically and subscribe to its button-click event. When clicking on the button, we run two asynchronous operations. One of them uses a regular `await` operator, while the other uses the `ConfigureAwait` method with `false` as a parameter value. It explicitly instructs that we should not use captured synchronization contexts to run continuation code on it. Inside each operation, we measure the time they take to complete, and then, we display the respective time and ratios on the main screen.

As a result, we see that the regular `await` operator takes much more time to complete. This is because we post 100,000 continuation tasks on the UI thread, which uses its message loop to asynchronously work with those tasks. In this case, we do not need this code to run on the UI thread, since we do not access the UI components from the asynchronous operation; using `ConfigureAwait` with `false` will be a much more efficient solution.

There is one more thing worth noting. Try to run the program by just clicking on the button and waiting for the results. Now, do the same thing again, but this time, click on the button and try to drag the application window from side to side in a random manner. You will note that the code on the captured synchronization context becomes slower! This funny side effect perfectly illustrates how dangerous asynchronous programming is. It is very easy to experience a situation like this, and it would be almost impossible to debug it if you have never experienced such a behavior before.

To be fair, let's see the opposite scenario. In the preceding code snippet, inside the `Click` method, uncomment the commented line and comment out the line immediately preceding it. When running the application, we will get a multithreaded control access exception because the code that sets the `Label` control text will not be posted on the captured context, but it will be executed on a thread pool worker thread instead.

Working around the `async void` method

This recipe describes why `async void` methods are quite dangerous to use. You will learn in what situations it is acceptable to use this method and what to use instead, when possible.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe7`.

How to do it...

To learn how to work with the `async void` method, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static async Task AsyncTaskWithErrors()
{
    string result = await GetInfoAsync("AsyncTaskException", 2);
    WriteLine(result);
}

static async void AsyncVoidWithErrors()
{
    string result = await GetInfoAsync("AsyncVoidException", 2);
    WriteLine(result);
}

static async Task AsyncTask()
{
    string result = await GetInfoAsync("AsyncTask", 2);
    WriteLine(result);
}

static async void AsyncVoid()
{
    string result = await GetInfoAsync("AsyncVoid", 2);
    WriteLine(result);
}

static async Task<string> GetInfoAsync(string name, int seconds)
{
    await Task.Delay(TimeSpan.FromSeconds(seconds));
    if (name.Contains("Exception"))
        throw new Exception($"Boom from {name}!");
    return
}
```

```

    $"Task {name} is running on a thread id {CurrentThread.
ManagedThreadId}." +
    $" Is thread pool thread: {CurrentThread.IsThreadPoolThread}";
}

```

4. Add the following code snippet inside the Main method:

```

Task t = AsyncTask();
t.Wait();

AsyncVoid();
Sleep(TimeSpan.FromSeconds(3));

t = AsyncTaskWithErrors();
while(!t.IsFaulted)
{
    Sleep(TimeSpan.FromSeconds(1));
}
WriteLine(t.Exception);

//try
//{
//    AsyncVoidWithErrors();
//    Thread.Sleep(TimeSpan.FromSeconds(3));
//}
//catch (Exception ex)
//{
//    Console.WriteLine(ex);
//}

int[] numbers = {1, 2, 3, 4, 5};
Array.ForEach(numbers, async number => {
    await Task.Delay(TimeSpan.FromSeconds(1));
    if (number == 3) throw new Exception("Boom!");
    WriteLine(number);
});

ReadLine();

```

5. Run the program.

How it works...

When the program starts, we start two asynchronous operations by calling the two methods, `AsyncTask` and `AsyncVoid`. The first method returns a `Task` object, while the other returns nothing since it is declared `async void`. They both return immediately since they are asynchronous, but then, the first one can be easily monitored with the returned task status or just by calling the `Wait` method on it. The only way to wait for the second method to complete is to literally wait for some time because we have not declared any object that we can use to monitor the state of the asynchronous operation. Of course, it is possible to use some kind of shared state variable and set it from the `async void` method while checking it from the calling method, but it is better to just return a `Task` object instead.

The most dangerous part is exception handling. In case of the `async void` method, an exception will be posted to a current synchronization context; in our case, a thread pool. An unhandled exception on a thread pool will terminate the whole process. It is possible to intercept unhandled exceptions using the `AppDomain.UnhandledException` event, but there is no way to recover the process from there. To experience this, we should uncomment the `try/catch` block inside the `Main` method and then run the program.

Another fact about using `async void` lambda expressions is that they are compatible with the `Action` type, which is widely used in the standard .NET Framework class library. It is very easy to forget about exception handling inside this lambda expression, which will crash the program again. To see an example of this, uncomment the second commented-out block inside the `Main` method.

I strongly recommend using `async void` only in UI event handlers. In all other situations, use the methods that return `Task` instead.

Designing a custom awaitable type

This recipe shows you how to design a very basic awaitable type that is compatible with the `await` operator.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe8`.

How to do it...

To design a custom awaitable type, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static async Task AsynchronousProcessing()
{
    var sync = new CustomAwaitable(true);
    string result = await sync;
    WriteLine(result);

    var async = new CustomAwaitable(false);
    result = await async;

    WriteLine(result);
}

class CustomAwaitable
{
    public CustomAwaitable(bool completeSynchronously)
    {
        _completeSynchronously = completeSynchronously;
    }

    public CustomAwaiter GetAwaiter()
    {
        return new CustomAwaiter(_completeSynchronously);
    }

    private readonly bool _completeSynchronously;
}
```

```
class CustomAwaiter : INotifyCompletion
{
    private string _result = "Completed synchronously";
    private readonly bool _completeSynchronously;

    public bool IsCompleted => _completeSynchronously;

    public CustomAwaiter(bool completeSynchronously)
    {
        _completeSynchronously = completeSynchronously;
    }

    public string GetResult()
    {
        return _result;
    }

    public void OnCompleted(Action continuation)
    {
        ThreadPool.QueueUserWorkItem( state => {
            Sleep(TimeSpan.FromSeconds(1));
            _result = GetInfo();
            continuation?.Invoke();
        });
    }

    private string GetInfo()
    {
        return
            $"Task is running on a thread id {CurrentThread.
ManagedThreadId}." +
            $" Is thread pool thread: {CurrentThread.IsThreadPoolThread}";
    }
}
```

4. Add the following code snippet inside the Main method:

```
Task t = AsynchronousProcessing();
t.Wait();
```

5. Run the program.

How it works...

To be compatible with the `await` operator, a type should comply with a number of requirements that are stated in the C# language specification. If you have Visual Studio 2015 installed, you may find the specifications document inside the `C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC#\Specifications\1033` folder (assuming you have a 64-bit OS and used the default installation path).

In paragraph 7.7.7.1, we find a definition of awaitable expressions:

The task of an `await` expression is required to be awaitable. An expression `t` is awaitable if one of the following holds:

- ▶ *`t` is of compile time type `dynamic`*
- ▶ *`t` has an accessible instance or extension method called `GetAwaiter` with no parameters and no type parameters, and a return type `A` for which all of the following hold:*
 1. *`A` implements the interface `System.Runtime.CompilerServices.INotifyCompletion` (hereafter known as `INotifyCompletion` for brevity).*
 2. *`A` has an accessible, readable instance property `IsCompleted` of type `bool`.*
 3. *`A` has an accessible instance method `GetResult` with no parameters and no type parameters.*

This information is enough to get started. First, we define an awaitable type `CustomAwaitable` and implement the `GetAwaiter` method. This in turn returns an instance of the `CustomAwaiter` type. `CustomAwaiter` implements the `INotifyCompletion` interface, has the `IsCompleted` property of the type `bool`, and has the `GetResult` method, which returns a `string` type. Finally, we write a piece of code that creates two `CustomAwaitable` objects and awaits both of them.

Now, we should understand the way `await` expressions are evaluated. This time, the specifications have not been quoted to avoid unnecessary details. Basically, if the `IsCompleted` property returns `true`, we just call the `GetResult` method synchronously. This prevents us from allocating resources for asynchronous task execution if the operation has already been completed. We cover this scenario by providing the `completeSynchronously` parameter to the constructor method of the `CustomAwaitable` object.

Otherwise, we register a callback action to the `OnCompleted` method of `CustomAwaiter` and start the asynchronous operation. When it gets completed, it calls the provided callback, which will get the result by calling the `GetResult` method on the `CustomAwaiter` object.



This implementation has been used for educational purposes only. Whenever you write asynchronous functions, the most natural approach is to use the standard `Task` type. You should define your own awaitable type only if you have a solid reason why you cannot use `Task` and you know exactly what you are doing.

There are many other topics related to designing custom awaitable types, such as the `ICriticalNotifyCompletion` interface implementation and synchronization context propagation. After understanding the basics of how an awaitable type is designed, you will be able to use the C# language specification and other information sources to find out the details you need with ease. But I would like to emphasize that you should just use the `Task` type, unless you have a really good reason not to.

Using the dynamic type with await

This recipe shows you how to design a very basic type that is compatible with the `await` operator and the dynamic C# type.

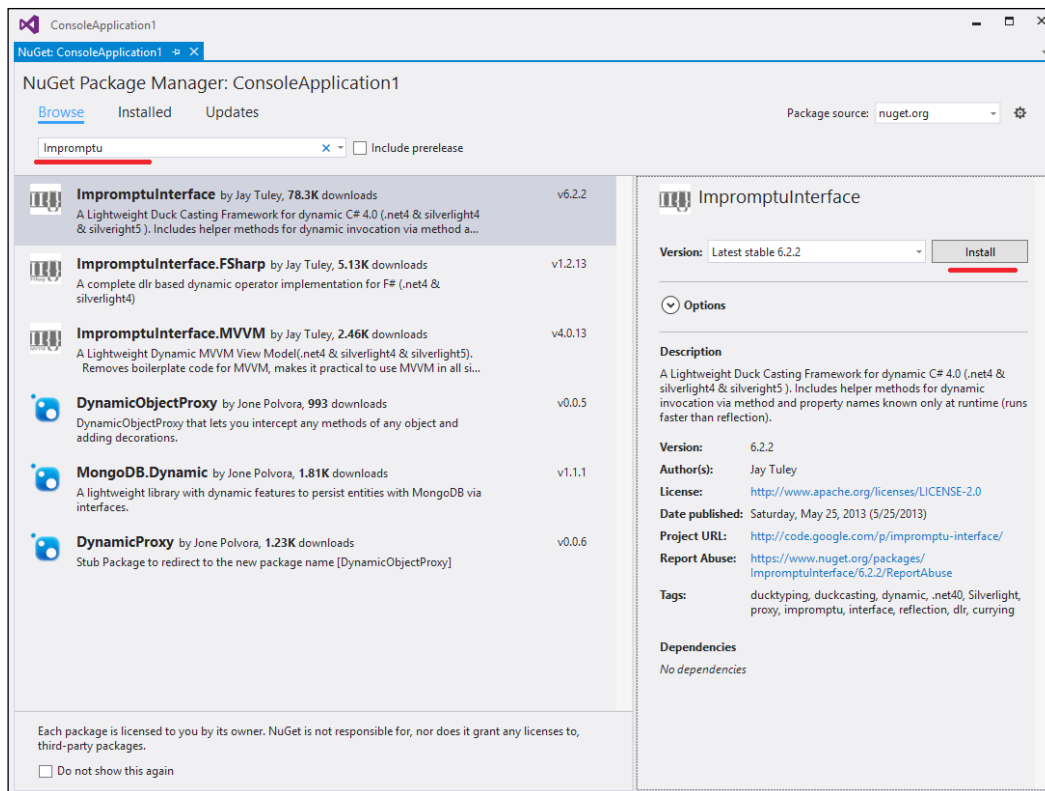
Getting ready

To step through this recipe, you will need Visual Studio 2015. You will need Internet access to download the NuGet package. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter5\Recipe9`.

How to do it...

To learn how to use the dynamic type with `await`, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add references to the **ImpromptuInterface** NuGet package by following these steps:
 1. Right-click on the **References** folder in the project, and select the **Manage NuGet Packages...** menu option.
 2. Now, add your preferred references to the **ImpromptuInterface NuGet** package. You can use the search function in the **Manage NuGet Packages** dialog as follows:



3. In the `Program.cs` file, use the following using directives:

```
using System;
using System.Dynamic;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using ImpromptuInterface;
using static System.Console;
using static System.Threading.Thread;
```

4. Add the following code snippet below the `Main` method:

```
static async Task AsynchronousProcessing()
{
    string result = await GetDynamicAwaitableObject(true);
    WriteLine(result);

    result = await GetDynamicAwaitableObject(false);
    WriteLine(result);
}
```



```
}

static dynamic GetDynamicAwaitableObject(bool
completeSynchronously)
{
    dynamic result = new ExpandoObject();
    dynamic awaiter = new ExpandoObject();

    awaiter.Message = "Completed synchronously";
    awaiter.IsCompleted = completeSynchronously;
    awaiter.GetResult = (Func<string>)(() => awaiter.Message);

    awaiter.OnCompleted = (Action<Action>) ( callback =>
        ThreadPool.QueueUserWorkItem(state => {
            Sleep(TimeSpan.FromSeconds(1));
            awaiter.Message = GetInfo();
            callback?.Invoke();
        })
    );

    IAwaiter<string> proxy = Impromptu.ActLike(awaiter);

    result.GetAwaiter = (Func<dynamic>) ( () => proxy );

    return result;
}

static string GetInfo()
{
    return
        $"Task is running on a thread id {CurrentThread.
ManagedThreadId}." +
        $" Is thread pool thread: {CurrentThread.IsThreadPoolThread}";
}
```

5. Add the following code below the Program class definition:

```
public interface IAwaiter<T> : INotifyCompletion
{
    bool IsCompleted { get; }

    T GetResult();
}
```

6. Add the following code snippet inside the `Main` method:

```
Task t = AsynchronousProcessing();  
t.Wait();
```

7. Run the program.

How it works...

Here, we repeat the trick from the previous recipe but this time, with the help of dynamic expressions. We can achieve this goal with the help of NuGet—a package manager that contains many useful libraries. This time, we use a library that dynamically creates wrappers, implementing the interfaces we need.

To start with, we create two instances of the `ExpandoObject` type and assign them to dynamic local variables. These variables will be our `awaitable` and `awaiter` objects. Since an `awaitable` object just requires having the `GetAwaiter` method, there are no problems with providing it. `ExpandoObject` (combined with the `dynamic` keyword) allows us to customize itself and add properties and methods by assigning corresponding values. It is in fact a dictionary-type collection with keys of the type `string` and values of the type `object`. If you are familiar with the JavaScript programming language, you might note that this is very similar to JavaScript objects.

Since `dynamic` allows us to skip compile-time checks in C#, `ExpandoObject` is written in such a way that if you assign something to a property, it creates a dictionary entry, where the key is the property name and a value is any value that is supplied. When you try to get the property value, it goes into the dictionary and provides the value that is stored in the corresponding dictionary entry. If the value is of the type `Action` or `Func`, we actually store a delegate, which in turn can be used like a method. Therefore, a combination of the `dynamic` type with `ExpandoObject` allows us to create an object and dynamically provide it with properties and methods.

Now, we need to construct our `awaiter` and `awaitable` objects. Let's start with `awaiter`. First, we provide a property called `Message` and an initial value to this property. Then, we define the `GetResult` method using a `Func<string>` type. We assign a lambda expression, which returns the `Message` property value. We then implement the `IsCompleted` property. If it is set to `true`, we can skip the rest of the work and proceed to our `awaitable` object that is stored in the `result` local variable. We just need to add a method returning the `dynamic` object and return our `awaiter` object from it. Then, we can use `result` as the `await` expression; however, it will run synchronously.

The main challenge is implementing asynchronous processing on our dynamic object. The C# language specifications state that an `awaiter` object must implement the `INotifyCompletion` or `ICriticalNotifyCompletion` interface, which `ExpandoObject` does not. And even when we implement the `OnCompleted` method dynamically, adding it to the `awaiter` object, we will not succeed because our object does not implement either of the aforementioned interfaces.

To work around this problem, we use the `ImpromptuInterface` library that we obtained from NuGet. It allows us to use the `Impromptu.ActLike` method to dynamically create proxy objects that will implement the required interface. If we try to create a proxy implementing the `INotifyCompletion` interface, we will still fail because the proxy object is not dynamic anymore, and this interface has the `OnCompleted` method only, but it does not have the `IsCompleted` property or the `GetResult` method. As the last workaround, we define a generic interface, `IAwaiter<T>`, which implements `INotifyCompletion` and adds all the required properties and methods. Now, we use it for proxy generation and change the `result` object to return a proxy instead of `awaiter` from the `GetAwaiter` method. The program now works; we just constructed an `awaitable` object that is completely dynamic at runtime.

6

Using Concurrent Collections

In this chapter, we will look through the different data structures for concurrent programming included in the .NET Framework base class library. You will learn the following recipes:

- ▶ Using `ConcurrentDictionary`
- ▶ Implementing asynchronous processing using `ConcurrentQueue`
- ▶ Changing the asynchronous processing order with `ConcurrentStack`
- ▶ Creating a scalable crawler with `ConcurrentBag`
- ▶ Generalizing asynchronous processing with `BlockingCollection`

Introduction

Programming requires understanding and knowledge of basic data structures and algorithms. To choose the best-suited data structure for a concurrent situation, a programmer has to know about many things, such as algorithm time, space complexity, and the big O notation. In different, well-known scenarios, we always know which data structures are more efficient.

For concurrent computations, we need to have appropriate data structures. These data structures have to be scalable, avoid locks when possible, and at the same time provide thread-safe access. .NET Framework, since version 4, has the `System.Collections.Concurrent` namespace with several data structures in it. In this chapter, we will cover several data structures and show you very simple examples of how to use them.

Let's start with `ConcurrentQueue`. This collection uses atomic **Compare and Swap (CAS)** operations, which allow us to safely exchange values of two variables, and `SpinWait` to ensure thread safety. It implements a **First In, First Out (FIFO)** collection, which means that the items go out of the queue in the same order in which they were added to the queue. To add an item to a queue, you call the `Enqueue` method. The `TryDequeue` method tries to take the first item from the queue, and the `TryPeek` method tries to get the first item without removing it from the queue.

The `ConcurrentStack` collection is also implemented without using any locks and only with CAS operations. This is the **Last In, First Out (LIFO)** collection, which means that the most recently added item will be returned first. To add items, you can use the `Push` and `PushRange` methods; to retrieve, you use `TryPop` and `TryPopRange`, and to inspect, you can use the `TryPeek` method.

The `ConcurrentBag` collection is an unordered collection that supports duplicate items. It is optimized for a scenario where multiple threads partition their work in such a way that each thread produces and consumes its own tasks, dealing with other threads' tasks very rarely (in which case, it uses locks). You add items to a bag using the `Add` method; you inspect with `TryPeek`, and take items from a bag with the `TryTake` method.



Avoid using the `Count` property on the collections mentioned. They are implemented using linked lists, and therefore, `Count` is an $O(N)$ operation. If you need to check whether the collection is empty, use the `IsEmpty` property, which is an $O(1)$ operation.

`ConcurrentDictionary` is a thread-safe dictionary collection implementation. It is lock-free for read operations. However, it requires locking for write operations. The concurrent dictionary uses multiple locks, implementing a fine-grained locking model over the dictionary buckets. The number of locks could be defined using a constructor with the `concurrencyLevel` parameter, which means that an estimated number of threads will update the dictionary concurrently.



Since a concurrent dictionary uses locking, there are a number of operations that require acquiring all the locks inside the dictionary. These operations are: `Count`, `IsEmpty`, `Keys`, `Values`, `CopyTo`, and `ToArray`. Avoid using these operations without need.

`BlockingCollection` is an advanced wrapper over the `IProducerConsumerCollection` generic interface implementation. It has many features that are more advanced and is very useful for implementing pipeline scenarios when you have some steps that use the results from processing the previous steps. The `BlockingCollection` class supports features such as blocking, bounding inner collections capacity, canceling collection operations, and retrieving values from multiple blocking collections.

The concurrent algorithms can be very complicated, and covering all the concurrent collections—whether more or less advanced—would require writing a separate book. Here, we illustrate only the simplest examples of using concurrent collections.

Using ConcurrentDictionary

This recipe shows you a very simple scenario, comparing the performance of a usual dictionary collection with the concurrent dictionary in a single-threaded environment.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter6\Recipe1`.

How to do it...

To understand the difference between the performance of a usual dictionary collection and the concurrent dictionary, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using static System.Console;
```

3. Add the following code snippet below the `Main` method:

```
const string Item = "Dictionary item";
const int Iterations = 1000000;
public static string CurrentItem;
```

4. Add the following code snippet inside the `Main` method:

```
var concurrentDictionary = new ConcurrentDictionary<int,
string>();
var dictionary = new Dictionary<int, string>();

var sw = new Stopwatch();

sw.Start();
for (int i = 0; i < Iterations; i++)
{
    lock (dictionary)
```

```
        {
            dictionary[i] = Item;
        }
    }
    sw.Stop();
    WriteLine($"Writing to dictionary with a lock: {sw.Elapsed}");

    sw.Restart();
    for (int i = 0; i < Iterations; i++)
    {
        concurrentDictionary[i] = Item;
    }
    sw.Stop();
    WriteLine($"Writing to a concurrent dictionary: {sw.Elapsed}");

    sw.Restart();
    for (int i = 0; i < Iterations; i++)
    {
        lock (dictionary)
        {
            CurrentItem = dictionary[i];
        }
    }
    sw.Stop();
    WriteLine($"Reading from dictionary with a lock: {sw.Elapsed}");

    sw.Restart();
    for (int i = 0; i < Iterations; i++)
    {
        CurrentItem = concurrentDictionary[i];
    }
    sw.Stop();
    WriteLine($"Reading from a concurrent dictionary: {sw.Elapsed}");
```

5. Run the program.

How it works...

When the program starts, we create two collections. One of them is a standard dictionary collection, and the other is a new concurrent dictionary. Then, we start adding to them, using a standard dictionary with a lock and measuring the time it takes for one million iterations to complete. Then, we measure the `ConcurrentDictionary` collection's performance in the same scenario, and we finally compare the performance of retrieving values from both collections.

In this very simple scenario, we find that `ConcurrentDictionary` is significantly slower on write operations than a usual dictionary with a lock but is faster on retrieval operations. Therefore, if we need many thread-safe reads from a dictionary, the `ConcurrentDictionary` collection is the best choice.



If you need just read-only multithreaded access to the dictionary, it may not be necessary to perform thread-safe reads. In this scenario, it is much better to use just a regular dictionary or the `ReadOnlyDictionary` collections.

The `ConcurrentDictionary` collection is implemented using the **fine-grained locking** technique, and this allows it to scale better on multiple writes than using a regular dictionary with a lock (which is called **coarse-grained locking**). As we saw in this example, when we use just one thread, a concurrent dictionary is much slower, but when we scale this up to five-six threads (if we have enough CPU cores that could run them simultaneously), the concurrent dictionary will actually perform better.

Implementing asynchronous processing using `ConcurrentQueue`

This recipe will show you an example of creating a set of tasks to be processed asynchronously by multiple workers.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter6\Recipe2`.

How to do it...

To understand the working of creating a set of tasks to be processed asynchronously by multiple workers, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
```


3. Add the following code snippet below the Main method:

```
static async Task RunProgram()
{
    var taskQueue = new ConcurrentQueue<CustomTask>();
    var cts = new CancellationTokenSource();

    var taskSource = Task.Run(() => TaskProducer(taskQueue));

    Task[] processors = new Task[4];
    for (int i = 1; i <= 4; i++)
    {
        string processorId = i.ToString();
        processors[i-1] = Task.Run(
            () => TaskProcessor(taskQueue, $"Processor {processorId}",
            cts.Token));
    }

    await taskSource;
    cts.CancelAfter(TimeSpan.FromSeconds(2));

    await Task.WhenAll(processors);
}

static async Task TaskProducer(ConcurrentQueue<CustomTask> queue)
{
    for (int i = 1; i <= 20; i++)
    {
        await Task.Delay(50);
        var workItem = new CustomTask {Id = i};
        queue.Enqueue(workItem);
        WriteLine($"Task {workItem.Id} has been posted");
    }
}

static async Task TaskProcessor(
    ConcurrentQueue<CustomTask> queue, string name,
    CancellationToken token)
{
    CustomTask workItem;
    bool dequeueSuccessful = false;

    await GetRandomDelay();
    do
    {
```

```

        dequeueSuccessful = queue.TryDequeue(out workItem);
        if (dequeueSuccessful)
        {
            WriteLine($"Task {workItem.Id} has been processed by
{name}");
        }

        await GetRandomDelay();
    }
    while (!token.IsCancellationRequested);
}

static Task GetRandomDelay()
{
    int delay = new Random(DateTime.Now.Millisecond).Next(1, 500);
    return Task.Delay(delay);
}

class CustomTask
{
    public int Id { get; set; }
}

```

4. Add the following code snippet inside the `Main` method:

```

Task t = RunProgram();
t.Wait();

```

5. Run the program.

How it works...

When the program runs, we create a queue of tasks with an instance of the `ConcurrentQueue` collection. Then, we create a cancellation token, which will be used to stop work after we are done posting tasks to the queue. Next, we start a separate worker thread that will post tasks to the tasks queue. This part produces a workload for our asynchronous processing.

Now, let's define a task-consuming part of the program. We create four workers that will wait a random time, get a task from the task queue, process it, and repeat the whole process until we signal the cancellation token. Finally, we start the task-producing thread, wait for its completion, and then signal the consumers that we've finished work with the cancellation token. The last step will be to wait for all our consumers to complete, to finish processing all tasks.

We see that we have tasks being processed from start to end, but it is possible that a later task will be processed before an earlier one because we have four workers running independently and the task processing time is not constant. We see that the access to the queue is thread-safe; no work item was taken twice.

Changing asynchronous processing order with ConcurrentStack

This recipe is a slight modification of the previous one. We will, once again, create a set of tasks to be processed asynchronously by multiple workers, but this time, we implement it with `ConcurrentStack` and see the differences.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter6\Recipe3`.

How to do it...

To understand the processing of a set of tasks implemented with `ConcurrentStack`, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
```

3. Add the following code snippet below the `Main` method:

```
static async Task RunProgram()
{
    var taskStack = new ConcurrentStack<CustomTask>();
    var cts = new CancellationTokenSource();

    var taskSource = Task.Run(() => TaskProducer(taskStack));

    Task[] processors = new Task[4];
    for (int i = 1; i <= 4; i++)
    {
        string processorId = i.ToString();
```

```

        processors[i - 1] = Task.Run(
            () => TaskProcessor(taskStack, $"Processor {processorId}",
            cts.Token));
    }

    await taskSource;
    cts.CancelAfter(TimeSpan.FromSeconds(2));

    await Task.WhenAll(processors);
}

static async Task TaskProducer(ConcurrentStack<CustomTask> stack)
{
    for (int i = 1; i <= 20; i++)
    {
        await Task.Delay(50);
        var workItem = new CustomTask { Id = i };
        stack.Push(workItem);
        WriteLine($"Task {workItem.Id} has been posted");
    }
}

static async Task TaskProcessor(
    ConcurrentStack<CustomTask> stack, string name,
    CancellationToken token)
{
    await GetRandomDelay();
    do
    {
        CustomTask workItem;
        bool popSuccessful = stack.TryPop(out workItem);
        if (popSuccessful)
        {
            WriteLine($"Task {workItem.Id} has been processed by
{name}");
        }

        await GetRandomDelay();
    }
    while (!token.IsCancellationRequested);
}

static Task GetRandomDelay()
{

```

```
        int delay = new Random(DateTime.Now.Millisecond).Next(1, 500);
        return Task.Delay(delay);
    }

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

4. Add the following code snippet inside the Main method:

```
Task t = RunProgram();
t.Wait();
```

5. Run the program.

How it works...

When the program runs, we now create an instance of the `ConcurrentStack` collection. The rest is almost like in the previous recipe, except instead of using the `Push` and `TryPop` methods on the concurrent stack, we use `Enqueue` and `TryDequeue` on a concurrent queue.

We now see that the task processing order has been changed. The stack is a LIFO collection, and workers process the latter tasks first. In case of a concurrent queue, tasks were processed in almost the same order in which they were added. This means that by depending on the number of workers, we will surely process the task that was created first in a given time frame. In the case of a stack, the tasks that were created earlier will have lower priority and may be not processed until a producer stops giving more tasks to the stack. This behavior is very specific and it is much better to use a queue in this scenario.

Creating a scalable crawler with ConcurrentBag

This recipe shows you how to scale workload between a number of independent workers that both produce work and process it.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter6\Recipe4`.

How to do it...

The following steps demonstrate how to scale workload between a number of independent workers that both produce work and process it:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Threading.Tasks;
using static System.Console;
```

3. Add the following code snippet below the `Main` method:

```
static Dictionary<string, string[]> _contentEmulation = new
Dictionary<string, string[]>();

static async Task RunProgram()
{
    var bag = new ConcurrentBag<CrawlingTask>();

    string[] urls = {"http://microsoft.com/", "http://google.com/",
"http://facebook.com/", "http://twitter.com/"};

    var crawlers = new Task[4];
    for (int i = 1; i <= 4; i++)
    {
        string crawlerName = $"Crawler {i}";
        bag.Add(new CrawlingTask { UrlToCrawl = urls[i-1],
ProducerName = "root" });
        crawlers[i - 1] = Task.Run(() => Crawl(bag, crawlerName));
    }

    await Task.WhenAll(crawlers);
}

static async Task Crawl(ConcurrentBag<CrawlingTask> bag, string
crawlerName)
{
    CrawlingTask task;
    while (bag.TryTake(out task))
    {
        IEnumerable<string> urls = await GetLinksFromContent(task);
        if (urls != null)
```

```
{
    foreach (var url in urls)
    {
        var t = new CrawlingTask
        {
            UrlToCrawl = url,
            ProducerName = crawlerName
        };

        bag.Add(t);
    }
}

WriteLine($"Indexing url {task.UrlToCrawl} posted by " +
    $"{task.ProducerName} is completed by {crawlerName}!");
}
}

static async Task<IEnumerable<string>> GetLinksFromContent (CrawlingTask task)
{
    await GetRandomDelay();

    if (_contentEmulation.ContainsKey(task.UrlToCrawl)) return _contentEmulation[task.UrlToCrawl];

    return null;
}

static void CreateLinks()
{
    _contentEmulation["http://microsoft.com/"] = new [] { "http://microsoft.com/a.html", "http://microsoft.com/b.html" };
    _contentEmulation["http://microsoft.com/a.html"] = new[] { "http://microsoft.com/c.html", "http://microsoft.com/d.html" };
    _contentEmulation["http://microsoft.com/b.html"] = new[] { "http://microsoft.com/e.html" };

    _contentEmulation["http://google.com/"] = new[] { "http://google.com/a.html", "http://google.com/b.html" };
    _contentEmulation["http://google.com/a.html"] = new[] { "http://google.com/c.html", "http://google.com/d.html" };
    _contentEmulation["http://google.com/b.html"] = new[] { "http://google.com/e.html", "http://google.com/f.html" };
    _contentEmulation["http://google.com/c.html"] = new[] { "http://google.com/h.html", "http://google.com/i.html" };
}
```

```

        _contentEmulation["http://facebook.com/"] = new [] { "http://facebook.com/a.html", "http://facebook.com/b.html" };
        _contentEmulation["http://facebook.com/a.html"] = new [] { "http://facebook.com/c.html", "http://facebook.com/d.html" };
        _contentEmulation["http://facebook.com/b.html"] = new [] { "http://facebook.com/e.html" };

        _contentEmulation["http://twitter.com/"] = new [] { "http://twitter.com/a.html", "http://twitter.com/b.html" };
        _contentEmulation["http://twitter.com/a.html"] = new [] { "http://twitter.com/c.html", "http://twitter.com/d.html" };
        _contentEmulation["http://twitter.com/b.html"] = new [] { "http://twitter.com/e.html" };
        _contentEmulation["http://twitter.com/c.html"] = new [] { "http://twitter.com/f.html", "http://twitter.com/g.html" };
        _contentEmulation["http://twitter.com/d.html"] = new [] { "http://twitter.com/h.html" };
        _contentEmulation["http://twitter.com/e.html"] = new [] { "http://twitter.com/i.html" };
    }

    static Task GetRandomDelay()
    {
        int delay = new Random(DateTime.Now.Millisecond).Next(150, 200);
        return Task.Delay(delay);
    }

    class CrawlingTask
    {
        public string UrlToCrawl { get; set; }

        public string ProducerName { get; set; }
    }

```

4. Add the following code snippet inside the Main method:

```

CreateLinks();
Task t = RunProgram();
t.Wait();

```

5. Run the program.

How it works...

The program simulates web page indexing with multiple web crawlers. A web crawler is a program that opens a web page by its address, indexes the content, tries to visit all the links that this page contains, and indexes these linked pages as well. At the beginning, we define a dictionary containing different web-page URLs. This dictionary simulates web pages containing links to other pages. The implementation is very naive; it does not care about indexing the already visited pages, but it is simple and allows us to focus on the concurrent workload.

Then, we create a concurrent bag, containing crawling tasks. We create four crawlers and provide a different site root URL to each of them. Then, we wait for all crawlers to compete. Now, each crawler starts to index the site URL it was given. We simulate the network I/O process by waiting for some random amount of time; then, if the page contains more URLs, the crawler posts more crawling tasks to the bag. Then, it checks whether there are any tasks left to crawl in the bag. If not, the crawler is complete.

If we check the output below the first four lines, which are root URLs, we will see that usually, which were root URLs, we will see that usually a task posted by the crawler number *N* is processed by the same crawler. However, the later lines will be different. This happens because internally, `ConcurrentBag` is optimized for exactly this scenario where there are multiple threads that both add items and remove them. This is achieved by letting each thread work with its own local queue of items, and thus, we do not need any locks while this queue is occupied. Only when we have no items left in the local queue will we perform some locking and try to *steal* the work from another thread's local queue. This behavior helps to distribute the work between all workers and avoid locking.

Generalizing asynchronous processing with BlockingCollection

This recipe will describe how to use `BlockingCollection` to simplify implementation of workload asynchronous processing.

Getting ready

To work through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter6\Recipe5`.

How to do it...

To understand how `BlockingCollection` simplifies the implementation of workload asynchronous processing, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Concurrent;
using System.Threading.Tasks;
using static System.Console;
```

3. Add the following code snippet below the `Main` method:

```
static async Task RunProgram(IProducerConsumerCollection<CustomTask> collection = null)
{
    var taskCollection = new BlockingCollection<CustomTask>();
    if(collection != null)
        taskCollection = new BlockingCollection<CustomTask>(collection);

    var taskSource = Task.Run(() => TaskProducer(taskCollection));

    Task[] processors = new Task[4];
    for (int i = 1; i <= 4; i++)
    {
        string processorId = $"Processor {i}";
        processors[i - 1] = Task.Run(
            () => TaskProcessor(taskCollection, processorId));
    }

    await taskSource;

    await Task.WhenAll(processors);
}

static async Task TaskProducer(BlockingCollection<CustomTask> collection)
{
    for (int i = 1; i <= 20; i++)
    {
        await Task.Delay(20);
        var workItem = new CustomTask { Id = i };
    }
}
```

```
        collection.Add(workItem);
        WriteLine($"Task {workItem.Id} has been posted");
    }
    collection.CompleteAdding();
}

static async Task TaskProcessor(
    BlockingCollection<CustomTask> collection, string name)
{
    await GetRandomDelay();
    foreach (CustomTask item in collection.GetConsumingEnumerable())
    {
        WriteLine($"Task {item.Id} has been processed by {name}");
        await GetRandomDelay();
    }
}

static Task GetRandomDelay()
{
    int delay = new Random(DateTime.Now.Millisecond).Next(1, 500);
    return Task.Delay(delay);
}

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

4. Add the following code snippet inside the Main method:

```
WriteLine("Using a Queue inside of BlockingCollection");
WriteLine();
Task t = RunProgram();
t.Wait();

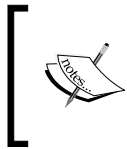
WriteLine();
WriteLine("Using a Stack inside of BlockingCollection");
WriteLine();
t = RunProgram(new ConcurrentStack<CustomTask>());
t.Wait();
```

5. Run the program.

How it works...

Here, we take exactly the first scenario, but now, we use a `BlockingCollection` class that provides many useful benefits. First of all, we are able to change the way the tasks are stored inside the blocking collection. By default, it uses a `ConcurrentQueue` container, but we are able to use any collection that implements the `IProducerConsumerCollection` generic interface. To illustrate this, we run the program twice, using `ConcurrentStack` as the underlying collection the second time.

Workers get work items by iterating the `GetConsumingEnumerable` method call result on a blocking collection. If there are no items inside the collection, the iterator will just block the worker thread until an item is posted to the collection. The cycle ends when the producer calls the `CompleteAdding` method on the collection. It signals that the work is done.



It is very easy to make a mistake and just iterate `BlockingCollection` as it implements `IEnumerable` itself. Do not forget to use `GetConsumingEnumerable`, or else, you will just iterate a "snapshot" of a collection and get completely unexpected program behavior.

The workload producer inserts the tasks into `BlockingCollection` and then calls the `CompleteAdding` method, which causes all the workers to get completed. Now, in the program output, we see two result sequences illustrating the difference between the concurrent queue and stack collections.

7

Using PLINQ

In this chapter, we will review different parallel programming paradigms, such as task and data parallelism, and cover the basics of data parallelism and parallel LINQ queries. You will learn the following recipes:

- ▶ Using the `Parallel` class
- ▶ Parallelizing a LINQ query
- ▶ Tweaking the parameters of a PLINQ query
- ▶ Handling exceptions in a PLINQ query
- ▶ Managing data partitioning in a PLINQ query
- ▶ Creating a custom aggregator for a PLINQ query

Introduction

In .NET Framework, there is a subset of libraries that is called Parallel Framework, often referred to as **Parallel Framework Extensions (PFX)**, which was the name of the very first version of these libraries. Parallel Framework was released with .NET Framework 4.0 and consists of three major parts:

- ▶ The Task Parallel Library (TPL)
- ▶ Concurrent collections
- ▶ Parallel LINQ or PLINQ

Until now, you have learned how to run several tasks in parallel and synchronize them with one another. In fact, we partitioned our program into a set of tasks and had different threads running different tasks. This approach is called **task parallelism**, and you have only been learning about task parallelism so far.

Imagine that we have a program that performs some heavy calculations over a big set of data. The easiest way to parallelize this program is to partition this set of data into smaller chunks, run the calculations needed over these chunks of data in parallel, and then aggregate the results of these calculations. This programming model is called **data parallelism**.

Task parallelism has the lowest abstraction level. We define a program as a combination of tasks, explicitly defining how they are combined. A program composed in this way could be very complex and detailed. Parallel operations are defined in different places in this program, and as it grows, the program becomes harder to understand and maintain. This way of making the program parallel is called **unstructured parallelism**. It is the price we have to pay if we have complex parallelization logic.

However, when we have simpler program logic, we can try to offload more parallelization details to the PFX libraries and the C# compiler. For example, we could say, "I would like to run those three methods in parallel, and I do not care how exactly this parallelization happens; let the .NET infrastructure decide the details". This raises the abstraction level as we do not have to provide a detailed description of how exactly we are parallelizing this. This approach is referred to as **structured parallelism** since the parallelization is usually a sort of declaration and each case of parallelization is defined in exactly one place in the program.



There could be an impression that unstructured parallelism is bad practice and structured parallelism should be always used instead. I would like to emphasize that this is not true. Structured parallelism is indeed more maintainable, and preferred when possible, but it is a much less universal approach. In general, there are many situations when we simply are not able to use it, and it is perfectly OK to use TPL task parallelism in an unstructured manner.

TPL has a `Parallel` class, which provides APIs for structured parallelism. This is still a part of TPL, but we will review it in this chapter because it is a perfect example of transition from a lower abstraction level to a higher one. When we use the `Parallel` class APIs, we do not need to provide the details of how we partition our work. However, we still need to explicitly define how we make one single result from partitioned results.

PLINQ has the highest abstraction level. It automatically partitions data in to chunks and decides whether we really need to parallelize the query or whether it will be more effective to use usual sequential query processing. Then, the PLINQ infrastructure takes care of combining the partitioned results. There are many options that programmers may tweak to optimize the query and achieve the best possible performance and result.

In this chapter, we will cover the `Parallel` class API usage and many different PLINQ options, such as making a LINQ query parallel, setting up an execution mode and tweaking the parallelism degree of a PLINQ query, dealing with a query item order, and handling PLINQ exceptions. You will also learn how to manage data partitioning for PLINQ queries.

Using the Parallel class

This recipe shows you how to use the `Parallel` class APIs. You will learn how to invoke methods in parallel, how to perform parallel loops, and tweak parallelization mechanics.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter7\Recipe1`.

How to do it...

To invoke methods in parallel, perform parallel loops, and tweak parallelization mechanics using the `Parallel` class, perform the given steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static string EmulateProcessing(string taskName)
{
    Sleep(TimeSpan.FromMilliseconds(
        new Random(DateTime.Now.Millisecond).Next(250, 350)));
    WriteLine($"{taskName} task was processed on a " +
        $"{Thread id {CurrentThread.ManagedThreadId}}");
    return taskName;
}
```

4. Add the following code snippet inside the `Main` method:

```
Parallel.Invoke(
    () => EmulateProcessing("Task1"),
    () => EmulateProcessing("Task2"),
    () => EmulateProcessing("Task3")
);

var cts = new CancellationTokensource();
```



```
var result = Parallel.ForEach(
    Enumerable.Range(1, 30),
    new ParallelOptions
    {
        CancellationTokens = cts.Token,
        MaxDegreeOfParallelism = Environment.ProcessorCount,
        TaskScheduler = TaskScheduler.Default
    },
    (i, state) =>
    {
        WriteLine(i);
        if (i == 20)
        {
            state.Break();
            WriteLine($"Loop is stopped: {state.IsStopped}");
        }
    });

WriteLine("---");
WriteLine($"IsCompleted: {result.IsCompleted}");
WriteLine($"Lowest break iteration: {result.
LowestBreakIteration}");
```

5. Run the program.

How it works...

This program demonstrates different features of the `Parallel` class. The `Invoke` method allows us to run several actions in parallel without much trouble as compared to defining tasks in TPL. The `Invoke` method blocks the other thread until all actions are complete, which is quite a common and convenient scenario.

The next feature is parallel loops, which are defined with the `For` and `ForEach` methods. We will look closely at `ForEach` since it is very similar to `For`. With the `ForEach` parallel loop, you can process any `IEnumerable` collection in parallel by applying an action delegate to each collection item. We are able to provide several options, customizing parallelization behavior, and get a result that shows whether the loop completed successfully.

To tweak our parallel loop, we provide an instance of the `ParallelOptions` class to the `ForEach` method. This allows us to cancel the loop with `CancellationToken`, restrict the maximum parallelism degree (how many maximum operations can be run in parallel), and provide a custom `TaskScheduler` class to schedule action tasks with it. Actions can accept an additional `ParallelLoopState` parameter, which is useful for breaking the loop or for checking what happens with the loop at this moment.

There are two ways of stopping the parallel loop with this state. We could use either the `Break` or `Stop` methods. The `Stop` method tells the loop to stop processing any more work and sets the `IsStopped` property of the parallel loop state to `true`. The `Break` method stops the iterations after it, but the initial ones will continue to work. In that case, the `LowestBreakIteration` property of the loop result will contain the number of lowest loop iteration where the `Break` method was called.

Parallelizing a LINQ query

This recipe will describe how to use PLINQ to make a query parallel and how to go back from a parallel query to sequential processing.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter7\Recipe2`.

How to do it...

To use PLINQ in order to make a query parallel and to go back from a parallel query to sequential processing, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void PrintInfo(string typeName)
{
    Sleep(TimeSpan.FromMilliseconds(150));
    WriteLine($"{typeName} type was printed on a thread " +
        $"id {CurrentThread.ManagedThreadId}");
}

static string EmulateProcessing(string typeName)
{
    Sleep(TimeSpan.FromMilliseconds(150));
```

```
        WriteLine($"{typeName} type was processed on a thread " +
            $"id {CurrentThread.ManagedThreadId}");
        return typeName;
    }

    static IEnumerable<string> GetTypes()
    {
        return from assembly in AppDomain.CurrentDomain.GetAssemblies()
               from type in assembly.GetExportedTypes()
               where type.Name.StartsWith("Web")
               select type.Name;
    }
```

4. Add the following code snippet inside the Main method:

```
var sw = new Stopwatch();
sw.Start();
var query = from t in GetTypes()
            select EmulateProcessing(t);

foreach (string typeName in query)
{
    PrintInfo(typeName);
}
sw.Stop();
WriteLine("---");
WriteLine("Sequential LINQ query.");
WriteLine($"Time elapsed: {sw.Elapsed}");
WriteLine("Press ENTER to continue....");
ReadLine();
Clear();
sw.Reset();

sw.Start();
var parallelQuery = from t in GetTypes().AsParallel()
                   select EmulateProcessing(t);

foreach (var typeName in parallelQuery)
{
    PrintInfo(typeName);
}
sw.Stop();
WriteLine("---");
```

```
WriteLine("Parallel LINQ query. The results are being merged on a
single thread");
WriteLine($"Time elapsed: {sw.Elapsed}");
WriteLine("Press ENTER to continue....");
ReadLine();
Clear();
sw.Reset();

sw.Start();
parallelQuery = from t in GetTypes().AsParallel()
                select EmulateProcessing(t);

parallelQuery.ForAll(PrintInfo);

sw.Stop();
WriteLine("---");
WriteLine("Parallel LINQ query. The results are being processed in
parallel");
WriteLine($"Time elapsed: {sw.Elapsed}");
WriteLine("Press ENTER to continue....");
ReadLine();
Clear();
sw.Reset();

sw.Start();
query = from t in GetTypes().AsParallel().AsSequential()
        select EmulateProcessing(t);

foreach (string typeName in query)
{
    PrintInfo(typeName);
}

sw.Stop();
WriteLine("---");
WriteLine("Parallel LINQ query, transformed into sequential.");
WriteLine($"Time elapsed: {sw.Elapsed}");
WriteLine("Press ENTER to continue....");
ReadLine();
Clear();
```

5. Run the program.

How it works...

When the program runs, we create a LINQ query that uses the reflection API to get all types whose names start with *Web* from the assemblies loaded in the current application domain. We emulate delays for processing each item and for printing it with the `EmulateProcessing` and `PrintInfo` methods. We also use the `Stopwatch` class to measure each query's execution time.

First, we run a usual sequential LINQ query. There is no parallelization here, so everything runs on the current thread. The second version of the query uses the `ParallelEnumerable` class explicitly. `ParallelEnumerable` contains the PLINQ logic implementation and is organized as a number of extension methods to the `IEnumerable` collection's functionality. Normally, we do not use this class explicitly; we are using it here to illustrate how PLINQ actually works. The second version runs `EmulateProcessing` in parallel; however, by default, the results are merged on a single thread, so the query execution time should be a couple of seconds less than the first version.

The third version shows how to use the `AsParallel` method to run the LINQ query in parallel in a declarative manner. We do not care about implementation details here but just state that we want to run this in parallel. However, the key difference in this version is that we use the `ForAll` method to print out the query results. It runs the action to all items in the query on the same thread they were processed in, skipping the results-merging step. It allows us to run `PrintInfo` in parallel as well, and this version runs even faster than the previous one.

The last sample shows how to turn a PLINQ query back to sequential with the `AsSequential` method. We can see that this query runs exactly like the first one.

Tweaking the parameters of a PLINQ query

This recipe shows how we can manage parallel processing options using a PLINQ query and what these options could affect during a query's execution.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter7\Recipe3`.

How to do it...

To understand how to manage parallel processing options using a PLINQ query and their effects, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static string EmulateProcessing(string typeName)
{
    Sleep(TimeSpan.FromMilliseconds(
        new Random(DateTime.Now.Millisecond).Next(250, 350)));
    WriteLine($"{typeName} type was processed on a thread " +
        $"id {CurrentThread.ManagedThreadId}");
    return typeName;
}

static IEnumerable<string> GetTypes()
{
    return from assembly in AppDomain.CurrentDomain.GetAssemblies()
           from type in assembly.GetExportedTypes()
           where type.Name.StartsWith("Web")
           orderby type.Name.Length
           select type.Name;
}
```

4. Add the following code snippet inside the `Main` method:

```
var parallelQuery = from t in GetTypes().AsParallel()
                    select EmulateProcessing(t);

var cts = new CancellationTokenSource();
cts.CancelAfter(TimeSpan.FromSeconds(3));

try
{
    parallelQuery
```

```
.WithDegreeOfParallelism(Environment.ProcessorCount)
.WithExecutionMode(ParallelExecutionMode.ForceParallelism)
.WithMergeOptions(ParallelMergeOptions.Default)
.WithCancellation(cts.Token)
.ForAll(WriteLine);
}
catch (OperationCanceledException)
{
    WriteLine("---");
    WriteLine("Operation has been canceled!");
}

WriteLine("---");
WriteLine("Unordered PLINQ query execution");
var unorderedQuery = from i in ParallelEnumerable.Range(1, 30)
    select i;

foreach (var i in unorderedQuery)
{
    WriteLine(i);
}

WriteLine("---");
WriteLine("Ordered PLINQ query execution");
var orderedQuery = from i in ParallelEnumerable.Range(1, 30).
    AsOrdered()
    select i;

foreach (var i in orderedQuery)
{
    WriteLine(i);
}
```

5. Run the program.

How it works...

The program demonstrates different useful PLINQ options that programmers can use. We start with creating a PLINQ query, and then we create another query providing PLINQ tweaking.

Let's start with cancelation first. To be able to cancel a PLINQ query, there is a `WithCancellation` method that accepts a cancelation token object. Here, we signal the cancelation token after 3 seconds, which leads to `OperationCanceledException` in the query and cancelation of the rest of the work.

Then, we are able to specify a parallelism degree for the query. It is the exact number of parallel partitions that will be used to execute the query. In the first recipe, we used the `Parallel.ForEach` loop, which has the maximum parallelism degree option. It is different because it specifies a maximum partitions value, but there could be fewer partitions if the infrastructure decides that it is better to use less parallelism to save resources and achieve optimal performance.

Another interesting option is overriding the query execution mode with the `WithExecutionMode` method. The PLINQ infrastructure can process some queries in sequential mode if it decides that parallelizing the query will only add more overhead and it actually will run slower. Using `WithExecutionMode`, we can force the query to run in parallel.

To tune up query result processing, we have the `WithMergeOptions` method. The default mode is used to buffer a number of results selected by the PLINQ infrastructure before returning them from the query. If the query takes a significant amount of time, it is more reasonable to turn off result buffering to get the results as soon as possible.

The last option is the `AsOrdered` method. It is possible that when we use parallel execution, the item order in the collection is not preserved. Later items in the collection could be processed before earlier ones. To prevent this, we need to call `AsOrdered` on a parallel query to explicitly tell the PLINQ infrastructure that we intend to preserve the item order for processing.

Handling exceptions in a PLINQ query

This recipe will describe how to handle exceptions in a PLINQ query.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter7\Recipe4`.

How to do it...

To understand how to handle exceptions in a PLINQ query, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;
```


3. Add the following code snippet inside the Main method:

```
IEnumerable<int> numbers = Enumerable.Range(-5, 10);

var query = from number in numbers
            select 100 / number;

try
{
    foreach(var n in query)
        WriteLine(n);
}
catch (DivideByZeroException)
{
    WriteLine("Divided by zero!");
}

WriteLine("---");
WriteLine("Sequential LINQ query processing");
WriteLine();

var parallelQuery = from number in numbers.AsParallel()
                   select 100 / number;

try
{
    parallelQuery.ForAll(WriteLine);
}
catch (DivideByZeroException)
{
    WriteLine("Divided by zero - usual exception handler!");
}
catch (AggregateException e)
{
    e.Flatten().Handle(ex =>
    {
        if (ex is DivideByZeroException)
        {
            WriteLine("Divided by zero - aggregate exception handler!");
            return true;
        }
    })
}
```

```

        return false;
    });
}

WriteLine("---");
WriteLine("Parallel LINQ query processing and results merging");

```

4. Run the program.

How it works...

First, we run a usual LINQ query over a range of numbers from -5 to 4. When we divide by 0, we get `DivideByZeroException`, and we handle it as usual in a `try/catch` block.

However, when we use `AsParallel`, we get `AggregateException` instead because we are now running in parallel, leveraging the task infrastructure behind the scenes. `AggregateException` will contain all the exceptions that occurred while running the PLINQ query. To handle the inner `DivideByZeroException` class, we use the `Flatten` and `Handle` methods, which were explained in the *Handling exceptions in asynchronous operations* recipe in *Chapter 5, Using C# 6.0*.



It is very easy to forget that when we handle aggregate exceptions, having more than one inner exception inside is a very common situation. If you forget to handle all of them, the exception will bubble up and the application will stop working.

Managing data partitioning in a PLINQ query

This recipe shows you how to create a very basic custom partitioning strategy to parallelize a LINQ query in a specific way.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter7\Recipe5`.

How to do it...

To learn how to create a very basic custom partitioning strategy to parallelize a LINQ query, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static void PrintInfo(string typeName)
{
    Sleep(TimeSpan.FromMilliseconds(150));
    WriteLine($"{typeName} type was printed on a thread " +
    $"id {CurrentThread.ManagedThreadId}");
}

static string EmulateProcessing(string typeName)
{
    Sleep(TimeSpan.FromMilliseconds(150));
    WriteLine($"{typeName} type was processed on a thread " +
    $"id { CurrentThread.ManagedThreadId}. Has " +
    $"{(typeName.Length % 2 == 0 ? "even" : "odd")} length.");

    return typeName;
}

static IEnumerable<string> GetTypes()
{
    var types = AppDomain.CurrentDomain
        .GetAssemblies()
        .SelectMany(a => a.GetExportedTypes());

    return from type in types
        where type.Name.StartsWith("Web")
        select type.Name;
}
```

```
public class StringPartitioner : Partitioner<string>
{
    private readonly IEnumerable<string> _data;

    public StringPartitioner(IEnumerable<string> data)
    {
        _data = data;
    }

    public override bool SupportsDynamicPartitions => false;

    public override IList<IEnumerator<string>>GetPartitions(
int partitionCount)
    {
        var result = new List<IEnumerator<string>>(
partitionCount);

        for (int i = 1; i <= partitionCount; i++)
        {
            result.Add(CreateEnumerator(i, partitionCount));
        }

        return result;
    }

    IEnumerator<string> CreateEnumerator(int partitionNumber, int
partitionCount)
    {
        int evenPartitions = partitionCount / 2;
        bool isEven = partitionNumber % 2 == 0;
        int step = isEven ? evenPartitions :
partitionCount - evenPartitions;

        int startIndex = partitionNumber / 2 +
partitionNumber % 2;

        var q = _data
            .Where(v => !(v.Length % 2 == 0 ^ isEven)
|| partitionCount == 1)
            .Skip(startIndex - 1);
```

```
        return q
            .Where((x, i) => i % step == 0)
            .GetEnumerator();
    }
}
```

4. Add the following code snippet inside the Main method:

```
var timer = Stopwatch.StartNew();
var partitioner = new StringPartitioner(GetTypes());
var parallelQuery = from t in partitioner.AsParallel()
//      .WithDegreeOfParallelism(1)
select EmulateProcessing(t);

parallelQuery.ForAll(PrintInfo);
int count = parallelQuery.Count();
timer.Stop();
WriteLine(" ----- ");
WriteLine($"Total items processed: {count}");
WriteLine($"Time elapsed: {timer.Elapsed}");
```

5. Run the program.

How it works...

To illustrate that we are able to choose custom partitioning strategies for the PLINQ query, we created a very simple partitioner that processes strings of odd and even lengths in parallel. To achieve this, we derive our custom `StringPartitioner` class from a standard base class `Partitioner<T>` using `string` as a type parameter.

We declare that we only support static partitioning by overriding the `SupportsDynamicPartitions` property and setting it to `false`. This means that we predefine our partitioning strategy. This is an easy way to partition the initial collection but could be inefficient depending on what data we have inside the collection. For example, in our case, if we had many strings with odd lengths and only one string with even length, one of the threads would have finished early and would not have helped to process odd-length strings. On the other hand, dynamic partitioning means that we partition the initial collection on the fly, balancing the work load between the worker threads.

Then, we implement the `GetPartitions` method, where we define the following logic: if there is only one partition, we simply process everything on it. However, if we have more than one partition, then we process strings with odd length on odd partitions and even-length strings on even-numbered partitions.



Please note that we need to create as many partitions as is stated in the `partitionCount` parameter, or else we will get the `Partitioner` returned a wrong number of partitions error.

Finally, we create an instance of our partitioner and perform a PLINQ query with it. We can see that different threads process the odd-length and even-length strings. Also, we can experiment with uncommenting the `WithDegreeOfParallelism` method and changing its parameter value. In the case of 1, there will be a sequential work items processing, and when increasing the value, we can see that more work gets done in parallel.

Creating a custom aggregator for a PLINQ query

This recipe shows you how to create a custom aggregation function for a PLINQ query.

Getting ready

To work through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter7\Recipe6`.

How to do it...

To understand the workings of a custom aggregation function for a PLINQ query, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static ConcurrentDictionary<char, int>
AccumulateLettersInformation(
    ConcurrentDictionary<char, int> taskTotal , string item)
{
    foreach (var c in item)
```

```
{
    if (taskTotal.ContainsKey(c))
    {
        taskTotal[c] = taskTotal[c] + 1;
    }
    else
    {
        taskTotal[c] = 1;
    }
}
WriteLine($"{item} type was aggregated on a thread " +
    $"id {CurrentThread.ManagedThreadId}");
return taskTotal;
}

static ConcurrentDictionary<char, int> MergeAccumulators(
    ConcurrentDictionary<char, int> total,
    ConcurrentDictionary<char, int> taskTotal)
{
    foreach (var key in taskTotal.Keys)
    {
        if (total.ContainsKey(key))
        {
            total[key] = total[key] + taskTotal[key];
        }
        else
        {
            total[key] = taskTotal[key];
        }
    }
    WriteLine("---");
    WriteLine($"Total aggregate value was calculated on a thread " +
        $"id {CurrentThread.ManagedThreadId}");
    return total;
}

static IEnumerable<string> GetTypes()
{
    var types = AppDomain.CurrentDomain
        .GetAssemblies()
        .SelectMany(a => a.GetExportedTypes());

    return from type in types
        where type.Name.StartsWith("Web")
        select type.Name;
}
```

4. Add the following code snippet inside the `Main` method:

```
var parallelQuery = from t in GetTypes().AsParallel()
                    select t;

var parallelAggregator = parallelQuery.Aggregate(
    () => new ConcurrentDictionary<char, int>(),
    (taskTotal, item) => AccumulateLettersInformation(taskTotal,
item),
    (total, taskTotal) => MergeAccumulators(total, taskTotal),
    total => total);

WriteLine();
WriteLine("There were the following letters in type names:");
var orderedKeys = from k in parallelAggregator.Keys
                  orderby parallelAggregator[k] descending
                  select k;

foreach (var c in orderedKeys)
{
    WriteLine($"Letter '{c}' ---- {parallelAggregator[c]} times");
}
```

5. Run the program.

How it works...

Here, we implement custom aggregation mechanics that are able to work with the PLINQ queries. To implement this, we have to understand that since a query is being processed in parallel by several tasks simultaneously, we need to provide mechanics to aggregate each task's result in parallel and then combine those aggregated values into one single result value.

In this recipe, we wrote an aggregating function that counts letters in a PLINQ query, which returns the `IEnumerable<string>` collection. It counts all the letters in each collection item. To illustrate the parallel aggregation process, we print out information about which thread processes each part of the aggregation.

We aggregate the PLINQ query results using the `Aggregate` extension method defined in the `ParallelEnumerable` class. It accepts four parameters, each of which is a function that performs different parts of the aggregation process. The first one is a factory that constructs the empty initial value of the aggregator. It is also called the seed value.



Note that the first value provided to the `Aggregate` method is actually not an initial seed value for the aggregator function but a factory method that constructs this initial seed value. If you provide just an instance, it will be used in all partitions that run in parallel, which will lead to an incorrect result.

The second function aggregates each collection item into the partition aggregation object. We implement this function with the `AccumulateLettersInformation` method. It iterates the string and counts the letters inside it. Here, the aggregation objects are different for each query partition running in parallel, which is why we called them `taskTotal`.

The third function is a higher level aggregation function that takes an aggregator object from a partition and merges it into a global aggregator object. We implement it with the `MergeAccumulators` method. The last function is a selector function that specifies what exact data we need from the global aggregator object.

Finally, we print out the aggregation result, ordering it by the letters used most often in the collection items.

8

Reactive Extensions

In this chapter, we will look at another interesting .NET library that helps us create asynchronous programs, **Reactive Extensions (Rx)**. We will cover the following recipes:

- ▶ Converting a collection to asynchronous `Observable`
- ▶ Writing a custom `Observable`
- ▶ Using the `Subject` type
- ▶ Creating an `Observable` object
- ▶ Using LINQ queries against an `Observable` collection
- ▶ Creating asynchronous operations with Rx

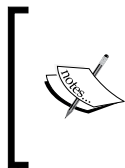
Introduction

As you have already learned, there are several approaches to creating asynchronous programs in .NET and C#. One of them is event-based asynchronous pattern, which has already been mentioned in the previous chapters. The initial goal of introducing events was to simplify the implementation of the `Observer` design pattern. This pattern is common for implementing notifications between objects.

When we discussed the Task Parallel Library, we noted that the event's main shortcoming was their inability to be effectively composed with each other. The other drawback was that the Event-based Asynchronous Pattern was not supposed to be used to deal with the sequence of notifications. Imagine that we have `IEnumerable<string>` that gives us string values. However, when we iterate it, we do not know how much time one iteration will take. It could be slow, and if we use the regular `foreach` loop or other synchronous iteration constructs, we will block our thread until we have the next value. This situation is called the **pull-based** approach, when we as a client pull values from the producer.

The opposite approach is the **push-based** approach, when the producer notifies the client about new values. This allows to offload work to the producer, while the client is free to do anything else in the time it waits for another value. Therefore, the goal is to get something like the asynchronous version of `IEnumerable`, which produces a sequence of values and notifies the consumer about each item in the sequence, when the sequence is complete or when an exception is thrown.

.NET Framework starting from version 4.0 contains the definition of the `IObservable<out T>` and `IObserver<in T>` interfaces that together represent the asynchronous push-based collection and its client. They come from the library called Reactive Extensions (or simply Rx) that was created inside Microsoft to help us effectively compose the sequence of events and all other types of asynchronous programs using observable collections. The interfaces were included in .NET Framework, but their implementations and all other mechanics are still distributed separately in the Rx library.



Rx globally is a cross-platform library. There are libraries for .NET 3.5, Silverlight, and Windows Phone. It is also available in JavaScript, Ruby, and Python. It is also open source; you can find Reactive Extensions' source code for .NET on the CodePlex website and other implementations on GitHub.

The most amazing thing is that the observable collections are compatible with LINQ, and therefore, we are able to use declarative queries to transform and compose those collections in an asynchronous manner. This also makes it possible for us to use the extension methods to add functionalities to the Rx programs in the same way it is used in the usual LINQ providers. Reactive Extensions also supports transition from all asynchronous programming patterns (including the Asynchronous Programming Model, the Event-based Asynchronous Pattern, and the Task Parallel Library) to observable collections, and it supports its own way of running asynchronous operations, which is still quite similar to TPL.

The Reactive Extensions library is a very powerful and complex instrument, which is worthy of writing a separate book. In this chapter, I would like to review the most useful scenario, that is, how to work with asynchronous event sequences effectively. We will observe key types of the Reactive Extensions framework, learn to create sequences and manipulate them in different ways, and finally, check how we could use Reactive Extensions to run asynchronous operations and manage their options.

Converting a collection to an asynchronous Observable

This recipe walks you through the process of creating an observable collection from an `Enumerable` class and how to process it asynchronously.

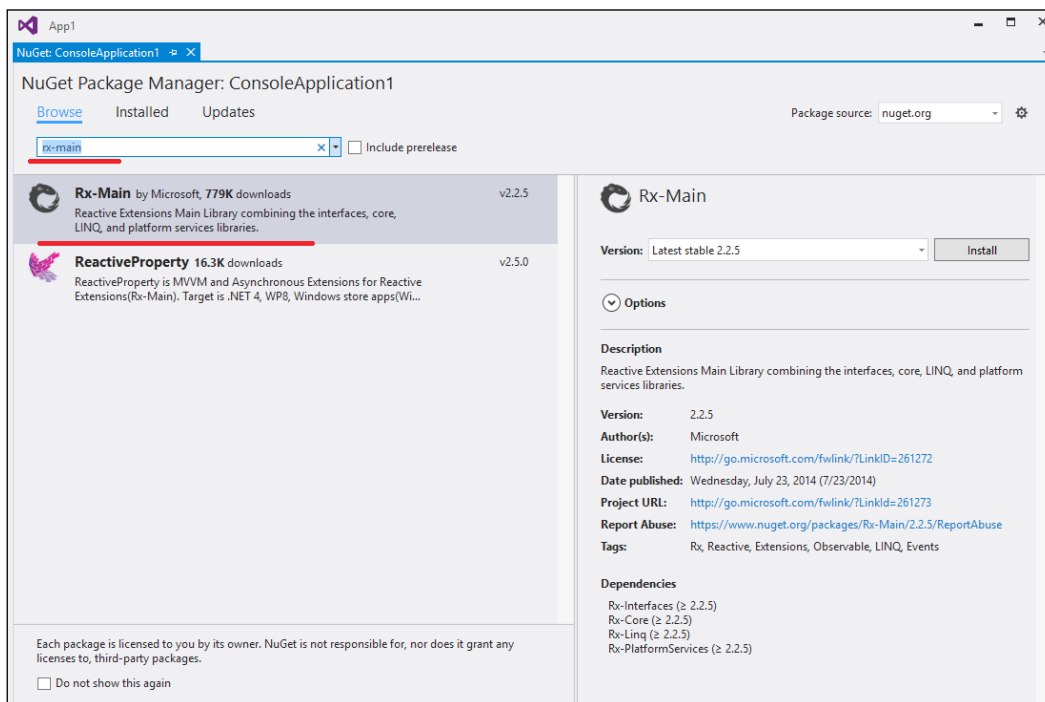
Getting ready

To work through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter8\Recipe1`.

How to do it...

To understand how to create an observable collection from an `Enumerable` class and process it asynchronously, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add a reference to the **Reactive Extensions Main Library** NuGet package by following these steps:
 1. Right-click on the **References** folder in the project, and select the **Manage NuGet Packages...** menu option.
 2. Now, add the **Reactive Extensions - Main Library** NuGet package. You can search for **rx-main** in the **Manage NuGet Packages** dialog, as shown in the following screenshot:



3. In the `Program.cs` file, add the following using directives:

```
using System;
using System.Collections.Generic;
using System.Reactive.Concurrency;
using System.Reactive.Linq;
using System.Threading;
using static System.Console;
using static System.Threading.Thread;
```

4. Add the following code snippet below the `Main` method:

```
static IEnumerable<int> EnumerableEventSequence()
{
    for (int i = 0; i < 10; i++)
    {
        Sleep(TimeSpan.FromSeconds(0.5));
        yield return i;
    }
}
```

5. Add the following code snippet inside the `Main` method:

```
foreach (int i in EnumerableEventSequence())
{
    Write(i);
}

WriteLine();
WriteLine("IEnumerable");

IObservable<int> o = EnumerableEventSequence().().ToObservable();
using (IDisposable subscription = o.Subscribe(Write))
{
    WriteLine();
    WriteLine("IObservable");
}

o = EnumerableEventSequence().ToObservable()
    .SubscribeOn(TaskPoolScheduler.Default);
using (IDisposable subscription = o.Subscribe(Write))
{
    WriteLine();
    WriteLine("IObservable async");
    ReadLine();
}
```

6. Run the program.

How it works...

Here, we simulate a slow enumerable collection with the `EnumerableEventSequence` method. Then, we iterate it with the usual `foreach` cycle, and we can see that it is actually slow; we wait for each iteration to complete.

We then convert this enumerable collection to `Observable` with the help of the `ToObservable` extension method from the Reactive Extensions library. Next, we subscribe to the updates of this observable collection, providing the `Console.WriteLine` method as the action, which will be executed on each update of the collection. As a result, we get exactly the same behavior as before; we wait for each iteration to complete because we use the main thread to subscribe to the updates.



We wrap the subscription objects into using statements. Although it is not always necessary, disposing off the subscriptions is a good practice that will help you avoid lifetime-related bugs.

To make the program asynchronous, we use the `SubscribeOn` method, providing it with the TPL task pool scheduler. This scheduler will place the subscription to the TPL task pool, offloading the work from the main thread. This allows us to keep the UI responsive and do something else while the collection gets updated. To check this behavior, you could remove the last `Console.ReadLine` call from the code. When doing so, we finish our main thread immediately, which forces all background threads (including the TPL task pool worker threads) to end as well, and we will get no output from the asynchronous collection.

If we are using a UI framework, we have to interact with the UI controls only from within the UI thread. To achieve this, we should use the `ObserveOn` method with the corresponding scheduler. For Windows Presentation Foundation, we have the `DispatcherScheduler` class and the `ObserveOnDispatcher` extension method defined in a separate NuGet package named `Rx-XAML` or `Reactive Extensions XAML support library`. For other platforms, there are corresponding separate NuGet packages as well.

Writing custom Observable

This recipe will describe how to implement the `IObservable<in T>` and `IObserver<out T>` interfaces to get the custom `Observable` sequence and properly consume it.

Getting ready

To step through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter8\Recipe2`.

How to do it...

To understand how to implement the `IObservable<in T>` and `IObserver<out T>` interfaces to get the custom Observable sequence and consume it, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add a reference to the **Reactive Extensions Main Library** NuGet package. Refer to the *Converting a collection to asynchronous observable* recipe for more details on how to do this.
3. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Generic;
using System.Reactive.Concurrency;
using System.Reactive.Disposables;
using System.Reactive.Linq;
using static System.Console;
using static System.Threading.Thread;
```

4. Add the following code snippet below the `Main` method:

```
class CustomObserver : IObserver<int>
{
    public void OnNext(int value)
    {
        WriteLine($"Next value: {value}; Thread Id: {CurrentThread.
ManagedThreadId}");
    }

    public void OnError(Exception error)
    {
        WriteLine($"Error: {error.Message}");
    }

    public void OnCompleted()
    {
        WriteLine("Completed");
    }
}

class CustomSequence : IObservable<int>
{
    private readonly IEnumerable<int> _numbers;
```

```
public CustomSequence(IEnumerable<int> numbers)
{
    _numbers = numbers;
}
public IDisposable Subscribe(IObserver<int> observer)
{
    foreach (var number in _numbers)
    {
        observer.OnNext(number);
    }
    observer.OnCompleted();
    return Disposable.Empty;
}
}
```

5. Add the following code snippet inside the Main method:

```
var observer = new CustomObserver();

var goodObservable = new CustomSequence(new[] {1, 2, 3, 4, 5});
var badObservable = new CustomSequence(null);

using (IDisposable subscription = goodObservable.
Subscribe(observer))
{
}

using (IDisposable subscription = goodObservable
.SubscribeOn(TaskPoolScheduler.Default).Subscribe(observer))
{
    Sleep(TimeSpan.FromMilliseconds(100));
    WriteLine("Press ENTER to continue");
    ReadLine();
}

using (IDisposable subscription = badObservable
.SubscribeOn(TaskPoolScheduler.Default).Subscribe(observer))
{
    Sleep(TimeSpan.FromMilliseconds(100));
    WriteLine("Press ENTER to continue");
    ReadLine();
}
```

6. Run the program.

How it works...

Here, we implement our observer first by simply printing out to the console the information about the next item from the observable collection, error, or sequence completion. This is a very simple consumer code and there is nothing special about it.

The interesting part is our observable collection implementation. We accept an enumeration of numbers into a constructor and do not check it for null on purpose. When we have a subscribing observer, we iterate this collection and notify the observer about each item in the enumeration.

Then, we demonstrate the actual subscription. As we can see, the asynchrony is achieved by calling the `SubscribeOn` method, which is an extension method to `IObservable` and contains asynchronous subscription logic. We do not care about asynchrony in our observable collection; we use standard implementation from the Reactive Extensions library.

When we subscribe to the normal observable collection, we just get all the items from it. It is now asynchronous, so we need to wait for some time for the asynchronous operation to complete and only then print the message and wait for the user input.

Finally, we try to subscribe to the next observable collection, where we are iterating a null enumeration and therefore getting a null reference exception. We see that the exception has been properly handled and the `OnError` method was executed to print out the error details.

Using the Subject type family

This recipe shows you how to use the `Subject` type family from the Reactive Extensions library.

Getting ready

To work through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter8\Recipe3`.

How to do it...

To understand the use of the `Subject` type family from the Reactive Extensions library, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add a reference to the **Reactive Extensions Main Library** NuGet package. Refer to the *Converting a collection to asynchronous observable* recipe for details on how to do this.

3. In the `Program.cs` file, add the following using directives:

```
using System;
using System.Reactive.Subjects;
using static System.Console;
using static System.Threading.Thread;
```

4. Add the following code snippet below the `Main` method:

```
static IDisposable OutputToConsole<T>(IObservable<T> sequence)
{
    return sequence.Subscribe(
        obj => WriteLine($"{obj}")
        , ex => WriteLine($"Error: {ex.Message}")
        , () => WriteLine("Completed")
    );
}
```

5. Add the following code snippet inside the `Main` method:

```
WriteLine("Subject");
var subject = new Subject<string>();

subject.OnNext("A");
using (var subscription = OutputToConsole(subject))
{
    subject.OnNext("B");
    subject.OnNext("C");
    subject.OnNext("D");
    subject.OnCompleted();
    subject.OnNext("Will not be printed out");
}

WriteLine("ReplaySubject");
var replaySubject = new ReplaySubject<string>();

replaySubject.OnNext("A");
using (var subscription = OutputToConsole(replaySubject))
{
    replaySubject.OnNext("B");
    replaySubject.OnNext("C");
    replaySubject.OnNext("D");
    replaySubject.OnCompleted();
}
```

```
WriteLine("Buffered ReplaySubject");
var bufferedSubject = new ReplaySubject<string>(2);

bufferedSubject.OnNext("A");
bufferedSubject.OnNext("B");
bufferedSubject.OnNext("C");
using (var subscription = OutputToConsole(bufferedSubject))
{
    bufferedSubject.OnNext("D");
    bufferedSubject.OnCompleted();
}

WriteLine("Time window ReplaySubject");
var timeSubject = new ReplaySubject<string>(TimeSpan.
FromMilliseconds(200));

timeSubject.OnNext("A");
Sleep(TimeSpan.FromMilliseconds(100));
timeSubject.OnNext("B");
Sleep(TimeSpan.FromMilliseconds(100));
timeSubject.OnNext("C");
Sleep(TimeSpan.FromMilliseconds(100));
using (var subscription = OutputToConsole(timeSubject))
{
    Sleep(TimeSpan.FromMilliseconds(300));
    timeSubject.OnNext("D");
    timeSubject.OnCompleted();
}

WriteLine("AsyncSubject");
var asyncSubject = new AsyncSubject<string>();

asyncSubject.OnNext("A");
using (var subscription = OutputToConsole(asyncSubject))
{
    asyncSubject.OnNext("B");
    asyncSubject.OnNext("C");
    asyncSubject.OnNext("D");
    asyncSubject.OnCompleted();
}

WriteLine("BehaviorSubject");
var behaviorSubject = new BehaviorSubject<string>("Default");
using (var subscription = OutputToConsole(behaviorSubject))
{
    behaviorSubject.OnNext("B");
}
```

```
behaviorSubject.OnNext("C");  
behaviorSubject.OnNext("D");  
behaviorSubject.OnCompleted();  
}
```

6. Run the program.

How it works...

In this program, we look through different variants of the `Subject` type family. The `Subject` type represents both the `IObservable` and `IObserver` implementations. This is useful in different proxy scenarios when we want to translate events from multiple sources to one stream, or vice versa, to broadcast an event sequence to multiple subscribers. Subjects are also very convenient for experimenting with Reactive Extensions.

Let's start with the basic `Subject` type. It retranslates an event sequence to subscribers as soon as they subscribe to it. In our case, the `A` string will not be printed out because the subscription happened after it was transmitted. Besides that, when we call the `OnCompleted` or `OnError` methods on `Observable`, it stops further translation of the event sequence, so the last string will also not be printed out.

The next type, `ReplaySubject`, is quite flexible and allows us to implement three additional scenarios. First, it can cache all the events from the beginning of their broadcasting, and if we subscribe later, we will get all the preceding events first. This behavior is illustrated in the second example. Here, we will have all four strings on the console because the first event will be cached and translated to the latter subscriber.

Then, we can specify the buffer size and the time window size for `ReplaySubject`. In the next example, we set the subject to have a buffer for two events. If more events are broadcasted, only the last two will be retranslated to the subscriber. So here, we will not see the first string because we have `B` and `C` in the subject buffer when subscribing to it. The same is the case with a time window. We can specify that the `Subject` type only caches events that took place less than a certain time ago, discarding the older ones. Therefore, in the fourth example, we will only see the last two events; the older events do not fit into the time window.

The `AsyncSubject` type is something like a `Task` type from the TPL globally. It represents a single asynchronous operation. If there are several events published, it waits for the event sequence completion and provides only the last event to the subscriber.

The `BehaviorSubject` type is quite similar to the `ReplaySubject` type, but it caches only one value and allows us to specify a default value in case we did not send any notifications. In our last example, we will see all the strings printed out because we provided a default value, and all other events take place after the subscription. If we move the `behaviorSubject.OnNext("B");` line upwards below the `Default` event, it will replace the default value in the output.

Creating an Observable object

This recipe will describe different ways to create an `Observable` object.

Getting ready

To work through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe could be found at `BookSamples\Chapter8\Recipe4`.

How to do it...

To understand different ways of creating an `Observable` object, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add a reference to the **Reactive Extensions Main Library** NuGet package. Refer to the *Converting a collection to asynchronous Observable* recipe for details on how to do this.
3. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Reactive.Disposables;
using System.Reactive.Linq;
using static System.Console;
using static System.Threading.Thread;
```

4. Add the following code snippet below the `Main` method:

```
static IDisposable OutputToConsole<T>(IObservable<T> sequence)
{
    return sequence.Subscribe(
        obj => WriteLine("{0}", obj)
        , ex => WriteLine("Error: {0}", ex.Message)
        , () => WriteLine("Completed")
    );
}
```

5. Add the following code snippet inside the `Main` method:

```
IObservable<int> o = Observable.Return(0);
using (var sub = OutputToConsole(o));
WriteLine(" ----- ");

o = Observable.Empty<int>();
using (var sub = OutputToConsole(o));
```

```

WriteLine(" ----- ");

o = Observable.Throw<int>(new Exception());
using (var sub = OutputToConsole(o));
WriteLine(" ----- ");

o = Observable.Repeat(42);
using (var sub = OutputToConsole(o.Take(5)));
WriteLine(" ----- ");

o = Observable.Range(0, 10);
using (var sub = OutputToConsole(o));
WriteLine(" ----- ");

o = Observable.Create<int>(ob => {
    for (int i = 0; i < 10; i++)
    {
        ob.OnNext(i);
    }
    return Disposable.Empty;
});
using (var sub = OutputToConsole(o));
WriteLine(" ----- ");

o = Observable.Generate(
    0 // initial state
    , i => i < 5 // while this is true we continue the sequence
    , i => ++i // iteration
    , i => i*2 // selecting result
);
using (var sub = OutputToConsole(o));
WriteLine(" ----- ");

IObservable<long> ol = Observable.Interval(TimeSpan.
FromSeconds(1));
using (var sub = OutputToConsole(ol))
{
    Sleep(TimeSpan.FromSeconds(3));
};
WriteLine(" ----- ");

ol = Observable.Timer(DateTimeOffset.Now.AddSeconds(2));
using (var sub = OutputToConsole(ol))
{
    Sleep(TimeSpan.FromSeconds(3));
};
WriteLine(" ----- ");

```

6. Run the program.

How it works...

Here, we walk through different scenarios of creating observable objects. Most of this functionality is provided as static factory methods of the `Observable` type. The first two samples show how we can create an `Observable` method that produces a single value and one that produces no value. In the next example, we use `Observable.Throw` to construct an `Observable` class that triggers the `OnError` handler of its observers.

The `Observable.Repeat` method represents an endless sequence. There are different overloads of this method; here, we construct an endless sequence by repeating 42 values. Then, we use LINQ's `Take` method to take five elements from this sequence. `Observable.Range` represents a range of values, pretty much like `Enumerable.Range`.

The `Observable.Create` method supports more custom scenarios. There are a lot of overloads that allow us to use cancellation tokens and tasks, but let's look at the simplest one. It accepts a function, which accepts an instance of observer and returns an `IDisposable` object representing a subscription. If we had any resources to clean up, we would be able to provide the cleanup logic here, but we just return an empty disposable as we actually do not need it.

The `Observable.Generate` method is another way to create a custom sequence. We must provide an initial value for a sequence and then a predicate that determines whether we should generate more items or complete the sequence. Then, we provide an iteration logic, which increments a counter in our case. The last parameter is a selector function that allows us to customize the results.

The last two methods deal with timers. `Observable.Interval` starts producing timer tick events with the `TimeSpan` period, and `Observable.Timer` specifies the startup time as well.

Using LINQ queries against an observable collection

This recipe shows you how to use LINQ to query an asynchronous sequence of events.

Getting ready

To work through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter8\Recipe5`.

How to do it...

To understand the use of LINQ queries against the observable collection, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add a reference to the **Reactive Extensions Main Library** NuGet package. Refer to the *Converting a collection to asynchronous observable* recipe for details on how to do this.
3. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Reactive.Linq;
using static System.Console;
```

4. Add the following code snippet below the `Main` method:

```
static IDisposable OutputToConsole<T>(IObservable<T> sequence, int
innerLevel)
{
    string delimiter = innerLevel == 0
        ? string.Empty
        : new string('-', innerLevel*3);

    return sequence.Subscribe(
        obj => WriteLine($"{delimiter}{obj}")
        , ex => WriteLine($"Error: {ex.Message}")
        , () => WriteLine($"{delimiter}Completed")
    );
}
```

5. Add the following code snippet inside the `Main` method:

```
IObservable<long> sequence = Observable.Interval(
    TimeSpan.FromMilliseconds(50)).Take(21);

var evenNumbers = from n in sequence
    where n % 2 == 0
    select n;

var oddNumbers = from n in sequence
    where n % 2 != 0
    select n;

var combine = from n in evenNumbers.Concat(oddNumbers)
    select n;
```



```
var nums = (from n in combine
            where n % 5 == 0
            select n)
            .Do(n => WriteLine($"-----Number {n} is processed in Do
method"));

using (var sub = OutputToConsole(sequence, 0))
using (var sub2 = OutputToConsole(combine, 1))
using (var sub3 = OutputToConsole(nums, 2))
{
    WriteLine("Press enter to finish the demo");
    ReadLine();
}
```

6. Run the program.

How it works...

The ability to use LINQ against the `Observable` event sequences is the main advantage of the Reactive Extensions framework. There are many different useful scenarios as well; unfortunately, it is impossible to show all of them here. I tried to provide a simple, yet very illustrative example, which does not have many complex details and shows the very essence of how a LINQ query could work when applied to asynchronous observable collections.

First, we create an `Observable` event that generates a sequence of numbers, one number every 50 milliseconds, and we start from the initial value of zero, taking 21 of those events. Then, we compose LINQ queries to this sequence. First, we select only the even numbers from the sequence, and then only the odd numbers. Then, we concatenate these two sequences.

The final query shows us how to use a very useful method, `Do`, which allows us to introduce side effects and, for example, logging each value from the resulting sequence. To run all queries, we create nested subscriptions, and because the sequences are initially asynchronous, we have to be very careful about the subscription's lifetime. The outer scope represents a subscription to the timer, and the inner subscriptions deal with the combined sequence query and the side effects query, respectively. If we press *Enter* too early, we just unsubscribe from the timer and thus stop the demo.

When we run the demo, we see the actual process of how different queries interact in real time. We can see that our queries are lazy, and they start running only when we subscribe to their results. The timer event's sequence is printed in the first column. When the even numbers query gets an even number, it prints it out as well using the `---` prefix to distinguish this sequence result from the first one. The final query results are printed in the right-hand column.

When the program runs, we can see that the timer sequence, the even-number sequence, and the side effect sequence run in parallel. Only the concatenation waits until the even-number sequence is complete. If we do not concatenate those sequences, we will have four parallel sequences of events interacting with each other in the most effective way! This shows the real power of Reactive Extensions and could be a good start to learn this library in depth.

Creating asynchronous operations with Rx

This recipe shows you how to create an `IObservable` from the asynchronous operations defined in other programming patterns.

Getting ready

To work through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter8\Recipe6`.

How to do it...

To understand how to create asynchronous operations with Rx, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add a reference to the **Reactive Extensions Main Library** NuGet package. Refer to the *Converting a collection to asynchronous observable* recipe for details on how to do this.
3. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Reactive;
using System.Reactive.Linq;
using System.Reactive.Threading.Tasks;
using System.Threading.Tasks;
using System.Timers;
using static System.Console;
using static System.Threading.Thread;
```

4. Add the following code snippet below the `Main` method:

```
static async Task<T> AwaitOnObservable<T>(IObservable<T>
observable)
{
    T obj = await observable;
    WriteLine($"{obj}");
    return obj;
}
```

```
static Task<string> LongRunningOperationTaskAsync(string name)
{
    return Task.Run(() => LongRunningOperation(name));
}

static IObservable<string> LongRunningOperationAsync(string name)
{
    return Observable.Start(() => LongRunningOperation(name));
}

static string LongRunningOperation(string name)
{
    Sleep(TimeSpan.FromSeconds(1));
    return $"Task {name} is completed. Thread Id {CurrentThread.
ManagedThreadId}";
}

static IDisposable OutputToConsole(IObservable<EventPattern<Elapse
dEventArgs>> sequence)
{
    return sequence.Subscribe(
        obj => WriteLine($"{obj.EventArgs.SignalTime}")
        , ex => WriteLine($"Error: {ex.Message}")
        , () => WriteLine("Completed")
    );
}

static IDisposable OutputToConsole<T>(IObservable<T> sequence)
{
    return sequence.Subscribe(
        obj => WriteLine("{0}", obj)
        , ex => WriteLine("Error: {0}", ex.Message)
        , () => WriteLine("Completed")
    );
}
```

5. Replace the Main method with the following code snippet:

```
delegate string AsyncDelegate(string name);

static void Main(string[] args)
{
    IObservable<string> o = LongRunningOperationAsync("Task1");
    using (var sub = OutputToConsole(o))
    {
        Sleep(TimeSpan.FromSeconds(2));
    };
}
```

```

WriteLine(" ----- ");

Task<string> t = LongRunningOperationTaskAsync("Task2");
using (var sub = OutputToConsole(t.ToObservable()))
{
    Sleep(TimeSpan.FromSeconds(2));
};
WriteLine(" ----- ");

AsyncDelegate asyncMethod = LongRunningOperation;

// marked as obsolete, use tasks instead
Func<string, IObservable<string>> observableFactory =
    Observable.FromAsyncPattern<string, string>(
        asyncMethod.BeginInvoke, asyncMethod.EndInvoke);

o = observableFactory("Task3");
using (var sub = OutputToConsole(o))
{
    Sleep(TimeSpan.FromSeconds(2));
};
WriteLine(" ----- ");

o = observableFactory("Task4");
AwaitOnObservable(o).Wait();
WriteLine(" ----- ");

using (var timer = new Timer(1000))
{
    var ot = Observable.
        FromEventPattern<ElapsedEventHandler,
        ElapsedEventArgs>(
            h => timer.Elapsed += h,
            h => timer.Elapsed -= h);
    timer.Start();


    using (var sub = OutputToConsole(ot))
    {
        Sleep(TimeSpan.FromSeconds(5));
    }
    WriteLine(" ----- ");
    timer.Stop();
}

```

6. Run the program.

How it works...

This recipe shows you how to convert different types of asynchronous operations to an `Observable` class. The first code snippet uses the `Observable.Start` method, which is quite similar to `Task.Run` from TPL. It starts an asynchronous operation that gives out a string result and then gets completed.

 I would strongly suggest that you use the Task Parallel Library for asynchronous operations. Reactive Extensions supports this scenario as well, but to avoid ambiguity, it is much better to stick with tasks when speaking about separate asynchronous operations and to go with Rx only when we need to work with sequences of events. Another suggestion is to convert every type of separate asynchronous operation to tasks and only then convert a task to an observable class, if you need it.

Then, we do the same with tasks and convert a task to an `Observable` method by simply calling the `ToObservable` extension method. The next code snippet is about converting the Asynchronous Programming Model pattern to `Observable`. Normally, you would convert APM to a task and then a task to `Observable`. However, there is a direct conversion, and this example illustrates how to run an asynchronous delegate and wrap it into an `Observable` operation.

The next part of the code snippet shows that we are able to use the `await` operator in an `Observable` operation. As we are not able to use the `async` modifier on an entry method such as `Main`, we introduce a separate method that returns a task and waits for this resulting task to be complete inside the `Main` method.

The last part of this code snippet is the same as the code which converts APM pattern to `Observable`, but now, we convert the Event-based Asynchronous Pattern directly to an `Observable` class. We create a timer and consume its events for 5 seconds. We then dispose the timer to clean up the resources.

9

Using Asynchronous I/O

In this chapter, we will review asynchronous I/O operations in detail. You will learn the following recipes:

- ▶ Working with files asynchronously
- ▶ Writing an asynchronous HTTP server and client
- ▶ Working with a database asynchronously
- ▶ Calling a WCF service asynchronously

Introduction

In the previous chapters, we already discussed how important it is to use asynchronous I/O operations properly. Why does it matter so much? To have a solid understanding, let's consider two kinds of applications.

When we run an application on a client, one of the most important things is to have a responsive user interface. This means that no matter what is happening with the application, all user interface elements, such as buttons and progress bars, keep running fast, and the user gets an immediate reaction from the application. This is not easy to achieve! If you try to open the Notepad text editor in Windows and try to load a text document that is several megabytes in size, the application window will be frozen for a significant amount of time because the whole text is being loaded from the disk first, and only then does the program start to process user input.

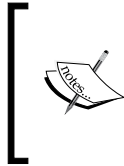
This is an extremely important issue, and in this situation, the only solution is to avoid blocking the UI thread at all costs. This in turn means that to prevent the blocking of the UI thread, every UI-related API must allow only asynchronous calls. This is the key reason behind redesigning APIs in the Windows 8 operating system by replacing almost every method with asynchronous analogs. But does it affect the performance if our application uses multiple threads to achieve this goal? Of course, it does! However, we could pay the price considering that we have only one user. It is good to have the application using all the power of the computer to be more effective, as all this power is intended for the single user who runs the application.

Let's look at the second case, then. If we run the application on a server, we have a completely different situation. We have scalability as a top priority, which means that a single user should consume as little resource as possible. If we start to create many threads for each user, we simply cannot scale well. It is a very complex problem to balance our application resource consumption in an efficient way. For example, in ASP.NET, which is a web application platform from Microsoft, we use a pool of worker threads to serve client requests. This pool has a limited number of worker threads, and we have to minimize the use time for each worker thread to achieve scalability. This means that we have to return it to the pool as soon as possible so that it can serve another request. If we start an asynchronous operation that requires computation, we will have a very inefficient workflow. First, we take a worker thread from the thread pool to serve a client request. Then, we take another worker thread and start an asynchronous operation on it. Now, we have two worker threads serving our request, but we really need the first thread to be doing something useful! Unfortunately, the common situation is that we simply wait for the asynchronous operation to complete, and we consume two worker threads instead of one. In this scenario, asynchrony is actually worse than synchronous execution! We do not need to load all the CPU cores as we are already serving many clients and thus are using all the CPU computing power. We do not need to keep the first thread responsive as we have no user interface. Then, why should we use asynchrony in server applications?

The answer is that we should use asynchrony when there is an asynchronous I/O operation. Today, modern computers usually have a hard disk drive that stores files and a network card that sends and receives data over the network. Both of these devices have their own microcomputers that manage I/O operations on a very low level and signal the operating system about the results. This is again quite a complicated topic; but to keep the concept clear, we could say that there is a way for programmers to start an I/O operation and provide the operating system with code to callback when the operation is completed. Between starting an I/O task and its completion, there is no CPU work involved; it is done in the corresponding disk and network controller microcomputers. This way of executing an I/O task is called an I/O thread; they are implemented using the .NET thread pool and in turn use an infrastructure from the operating system called I/O completion ports.

In ASP.NET, as soon as an asynchronous I/O operation is started from a worker thread, it can be returned immediately to the thread pool! While the operation is going on, this thread can serve other clients. Finally, when the operation signals completion, the ASP.NET infrastructure gets a free worker thread from the thread pool (which could be different from the one that started the operation), and it finishes the operation.

All right; we now understand how important I/O threads are for server applications. Unfortunately, it is very hard to check whether any given API uses I/O threads under the hood. The only way (besides studying the source code) is simply to know which .NET Framework class library leverages I/O threads. In this chapter, we will see how to use some of those APIs. You will learn how to work with files asynchronously, how to use network I/O to create an HTTP server and call the **Windows Communication Foundation (WCF)** service, and how to work with an asynchronous API to query a database.



Another important issue to consider is parallelism. For a number of reasons, an intensive parallel disk operation might have very poor performance. Be aware that parallel I/O operations are often very ineffective, and it might be reasonable to work with I/O sequentially, but in an asynchronous manner.

Working with files asynchronously

This recipe walks us through how to create a file and how to read and write data asynchronously.

Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter9\Recipe1`.

How to do it...

To understand how to work with files asynchronously, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;
using static System.Text.Encoding;
```

3. Add the following code snippet below the `Main` method:

```
const int BUFFER_SIZE = 4096;

static async Task ProcessAsynchronousIO()
{
```



```
using (var stream = new FileStream(
    "test1.txt", FileMode.Create, FileAccess.ReadWrite,
    FileShare.None, BUFFER_SIZE))
{
    WriteLine($"1. Uses I/O Threads: {stream.IsAsync}");

    byte[] buffer = UTF8.GetBytes(CreateFileContent());
    var writeTask = Task.Factory.FromAsync(
        stream.BeginWrite, stream.EndWrite, buffer, 0,
        buffer.Length, null);

    await writeTask;
}

using (var stream = new FileStream("test2.txt",
    FileMode.Create, FileAccess.ReadWrite, FileShare.None,
    BUFFER_SIZE, FileOptions.Asynchronous))
{
    WriteLine($"2. Uses I/O Threads: {stream.IsAsync}");

    byte[] buffer = UTF8.GetBytes(CreateFileContent());
    var writeTask = Task.Factory.FromAsync(
        stream.BeginWrite, stream.EndWrite, buffer, 0,
        buffer.Length, null);

    await writeTask;
}

using (var stream = File.Create("test3.txt", BUFFER_SIZE,
    FileOptions.Asynchronous))
using (var sw = new StreamWriter(stream))
{
    WriteLine($"3. Uses I/O Threads: {stream.IsAsync}");
    await sw.WriteAsync(CreateFileContent());
}

using (var sw = new StreamWriter("test4.txt", true))
{
    WriteLine($"4. Uses I/O Threads:
{((FileStream)sw.BaseStream).IsAsync}");
    await sw.WriteAsync(CreateFileContent());
}

WriteLine("Starting parsing files in parallel");
```

```

var readTasks = new Task<long>[4];
for (int i = 0; i < 4; i++)
{
    string fileName = $"test{i + 1}.txt";
    readTasks[i] = SumFileContent(fileName);
}

long[] sums = await Task.WhenAll(readTasks);

WriteLine($"Sum in all files: {sums.Sum()}");

WriteLine("Deleting files...");

Task[] deleteTasks = new Task[4];
for (int i = 0; i < 4; i++)
{
    string fileName = $"test{i + 1}.txt";
    deleteTasks[i] = SimulateAsynchronousDelete(fileName);
}

await Task.WhenAll(deleteTasks);

WriteLine("Deleting complete.");
}

static string CreateFileContent()
{
    var sb = new StringBuilder();
    for (int i = 0; i < 100000; i++)
    {
        sb.Append($"{{new Random(i).Next(0, 99999)}}");
        sb.AppendLine();
    }
    return sb.ToString();
}

static async Task<long> SumFileContent(string fileName)
{
    using (var stream = new FileStream(fileName,
        FileMode.Open, FileAccess.Read, FileShare.None, BUFFER_SIZE,
        FileOptions.Asynchronous))
    using (var sr = new StreamReader(stream))
    {
        long sum = 0;

```

```
        while (sr.Peek() > -1)
        {
            string line = await sr.ReadLineAsync();
            sum += long.Parse(line);
        }

        return sum;
    }

    static Task SimulateAsynchronousDelete(string fileName)
    {
        return Task.Run(() => File.Delete(fileName));
    }
```

4. Add the following code snippet inside the Main method:

```
var t = ProcessAsynchronousIO();
t.GetAwaiter().GetResult();
```

5. Run the program.

How it works...

When the program runs, we create four files in different ways and fill them up with random data. In the first case, we use the `FileStream` class and its methods, converting an Asynchronous Programming Model API to a task; in the second case, we do the same, but we provide `FileOptions.Asynchronous` to the `FileStream` constructor.



It is very important to use the `FileOptions.Asynchronous` option. If we omit this option, we can still work with the file in an asynchronous manner, but this is just an asynchronous delegate invocation on a thread pool! We use the I/O asynchrony with the `FileStream` class only if we provide this option (or bool `useAsync` in another constructor overload).

The third case uses some simplifying APIs, such as the `File.Create` method and the `StreamWriter` class. It still uses I/O threads, which we are able to check using the `stream.IsAsync` property. The last case illustrates that oversimplifying is also bad. Here, we do not leverage the I/O asynchrony by imitating it with the help of asynchronous delegate invocation.

Now, we perform parallel asynchronous reading from files, sum up their content, and then sum it with each other. Finally, we delete all the files. As there is no asynchronous delete file in any non-Windows store application, we simulate the asynchrony using the `Task.Run` factory method.

Writing an asynchronous HTTP server and client

This recipe shows you how to create a simple asynchronous HTTP server.

Getting ready

To step through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter9\Recipe2`.

How to do it...

The following steps demonstrate how to create a simple asynchronous HTTP server:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add a reference to the `System.Net.Http` framework library.
3. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.IO;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using static System.Console;
```

4. Add the following code snippet below the `Main` method:

```
static async Task GetResponseAsync(string url)
{
    using (var client = new HttpClient())
    {
        HttpResponseMessage responseMessage =
            await client.GetAsync(url);
        string responseHeaders = responseMessage.Headers.ToString();
        string response =
            await responseMessage.Content.ReadAsStringAsync();

        WriteLine("Response headers:");
        WriteLine(responseHeaders);
        WriteLine("Response body:");
        WriteLine(response);
    }
}
```

```
class AsyncHttpServer
{
    readonly HttpListener _listener;
    const string RESPONSE_TEMPLATE =
        "<html><head><title>Test</title></>
head><body><h2>Testpage</h2>" +
"<h4>Today is: {0}</h4></body></html>";

    public AsyncHttpServer(int portNumber)
    {
        _listener = new HttpListener();
        _listener.Prefixes.Add($"http://localhost:{portNumber}/");
    }

    public async Task Start()
    {
        _listener.Start();

        while (true)
        {
            var ctx = await _listener.GetContextAsync();
            WriteLine("Client connected...");
            var response = string.Format(RESPONSE_TEMPLATE,
                DateTime.Now);

            using (var sw = new StreamWriter(ctx.Response.OutputStream))
            {
                await sw.WriteAsync(response);
                await sw.FlushAsync();
            }
        }
    }

    public async Task Stop()
    {
        _listener.Abort();
    }
}
```

5. Add the following code snippet inside the Main method:

```
var server = new AsyncHttpServer(1234);
var t = Task.Run(() => server.Start());
WriteLine("Listening on port 1234. Open http://localhost:1234 in
your browser.");
```

```
WriteLine("Trying to connect:");  
WriteLine();  
  
GetResponseAsync("http://localhost:1234").GetAwaiter().  
GetResult();  
  
WriteLine();  
WriteLine("Press Enter to stop the server.");  
ReadLine();  
  
server.Stop().GetAwaiter().GetResult();
```

6. Run the program.

How it works...

Here, we implement a very simple web server using the `HttpListener` class. There is also a `TcpListener` class for the TCP socket I/O operations. We configure our listener to accept connections from any host to the local machine on port 1234. Then, we start the listener in a separate worker thread so that we can control it from the main thread.

The asynchronous I/O operation happens when we use the `GetContextAsync` method. Unfortunately, it does not accept `CancellationToken` for cancelation scenarios; so, when we want to stop the server, we just call the `_listener.Abort` method, which abandons the connection and stops the server.

To perform an asynchronous request on this server, we use the `HttpClient` class located in the `System.Net.Http` assembly and the same namespace. We use the `GetAsync` method to issue an asynchronous HTTP GET request. There are methods for other HTTP requests such as POST, DELETE, and PUT as well. `HttpClient` has many other options such as serializing and deserializing an object using different formats, such as XML and JSON, specifying a proxy server address, credentials, and so on.

When you run the program, you can see that the server has been started up. In the server code, we use the `GetContextAsync` method to accept new client connections. This method returns when a new client connects, and we simply output a very basic HTML language with the current date and time to the response. Then, we request the server and print the response headers and content. You can also open your browser and browse to `http://localhost:1234/`. Here, you will see the same response displayed in the browser window.

Working with a database asynchronously

This recipe walks us through the process of creating a database, populating it with data, and reading data asynchronously.

Getting ready

To step through this recipe, you will need Visual Studio 2015. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter9\Recipe3`.

How to do it...

To understand the process of creating a database, populating it with data, and reading data asynchronously, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.IO;
using System.Reflection;
using System.Threading.Tasks;
using static System.Console;
```

3. Add the following code snippet below the `Main` method:

```
static async Task ProcessAsynchronousIO(string dbName)
{
    try
    {
        const string connectionString =
            @"Data Source=(LocalDB)\MSSQLLocalDB;Initial
Catalog=master;" +
            "Integrated Security=True";

        string outputFolder = Path.GetDirectoryName(
            Assembly.GetExecutingAssembly().Location);

        string dbFileName = Path.Combine(outputFolder,
            $"{dbName}.mdf");
        string dbLogFileName = Path.Combine(outputFolder,
            $"{dbName}_log.ldf");
```

```

string dbConnectionString =
    @"Data Source=(LocalDB)\MSSQLLocalDB;" +
    $"AttachDBFileName={dbFileName};Integrated Security=True;";

using (var connection = new SqlConnection(connectionString))
{
    await connection.OpenAsync();

    if (File.Exists(dbFileName))
    {
        WriteLine("Detaching the database...");

        var detachCommand = new SqlCommand("sp_detach_db",
connection);
        detachCommand.CommandType = CommandType.StoredProcedure;
        detachCommand.Parameters.AddWithValue("@dbname", dbName);

        await detachCommand.ExecuteNonQuery();

        WriteLine("The database was detached succesfully.");
        WriteLine("Deleting the database...");

        if (File.Exists(dbLogFileName)) File.Delete(dbLogFileName);
        File.Delete(dbFileName);

        WriteLine("The database was deleted succesfully.");
    }

    WriteLine("Creating the database...");
    string createCommand =
        $"CREATE DATABASE {dbName} ON (NAME = N'{dbName}',
FILENAME = " +
        $"'{dbFileName}')";
    var cmd = new SqlCommand(createCommand, connection);

    await cmd.ExecuteNonQuery();
    WriteLine("The database was created succesfully");
}

using (var connection = new SqlConnection(dbConnectionString))
{
    await connection.OpenAsync();

```



```
var cmd = new SqlCommand("SELECT newid()", connection);
var result = await cmd.ExecuteScalarAsync();

WriteLine($"New GUID from DataBase: {result}");

cmd = new SqlCommand(
@"CREATE TABLE [dbo].[CustomTable] ( [ID] [int] IDENTITY(1,1) NOT
NULL, " +
"[Name] [nvarchar] (50) NOT NULL, CONSTRAINT [PK_ID] PRIMARY KEY
CLUSTERED " +
" ([ID] ASC) ON [PRIMARY]) ON [PRIMARY]", connection);

await cmd.ExecuteNonQueryAsync();

WriteLine("Table was created succesfully.");

cmd = new SqlCommand(
@"INSERT INTO [dbo].[CustomTable] (Name) VALUES ('John');
INSERT INTO [dbo].[CustomTable] (Name) VALUES ('Peter');
INSERT INTO [dbo].[CustomTable] (Name) VALUES ('James');
INSERT INTO [dbo].[CustomTable] (Name) VALUES ('Eugene');",
connection);
await cmd.ExecuteNonQueryAsync();

WriteLine("Inserted data succesfully");
WriteLine("Reading data from table...");

cmd = new SqlCommand(@"SELECT * FROM [dbo].[CustomTable]",
connection);
using (SqlDataReader reader = await cmd.
ExecuteReaderAsync())
{
    while (await reader.ReadAsync())
    {
        var id = reader.GetFieldValue<int>(0);
        var name = reader.GetFieldValue<string>(1);

        WriteLine("Table row: Id {0}, Name {1}", id, name);
    }
}
```

```

    }
}
catch(Exception ex)
{
    WriteLine("Error: {0}", ex.Message);
}
}

```

4. Add the following code snippet inside the Main method:

```

const string dataBaseName = "CustomDatabase";
var t = ProcessAsynchronousIO(dataBaseName);
t.GetAwaiter().GetResult();
Console.WriteLine("Press Enter to exit");
Console.ReadLine();

```

5. Run the program.

How it works...

This program works with software called SQL Server 2014 LocalDb. It is installed with Visual Studio 2015 and should work fine. However, in case of errors, you might want to repair this component from the installation wizard.

We start with configuring paths to our database files. We place database files in the program-execution folder. There will be two files: one for the database itself and another for the transaction log file. We also configure two connection strings that define how we connect to our databases. The first one is to connect to the LocalDb engine to detach our database; if it already exists, delete and then recreate it. We leverage the I/O asynchrony while opening the connection and while executing the SQL commands using the `OpenAsync` and `ExecuteNonQueryAsync` methods, respectively.

After this task is completed, we attach a newly created database. Here, we create a new table and insert some data in it. In addition to the previously mentioned methods, we use `ExecuteScalarAsync` to asynchronously get a scalar value from the database engine, and we use the `SqlDataReader.ReadAsync` method to read a data row from the database table asynchronously.

If we had a large table with large binary values in its rows in our database, then we would use the `CommandBehavior.SequentialAccess` enumeration to create the data reader and the `GetFieldValueAsync` method to get large field values from the reader asynchronously.

Calling a WCF service asynchronously

This recipe will describe how to create a WCF service, how to host it in a console application, how to make service metadata available to clients, and how to consume it in an asynchronous way.

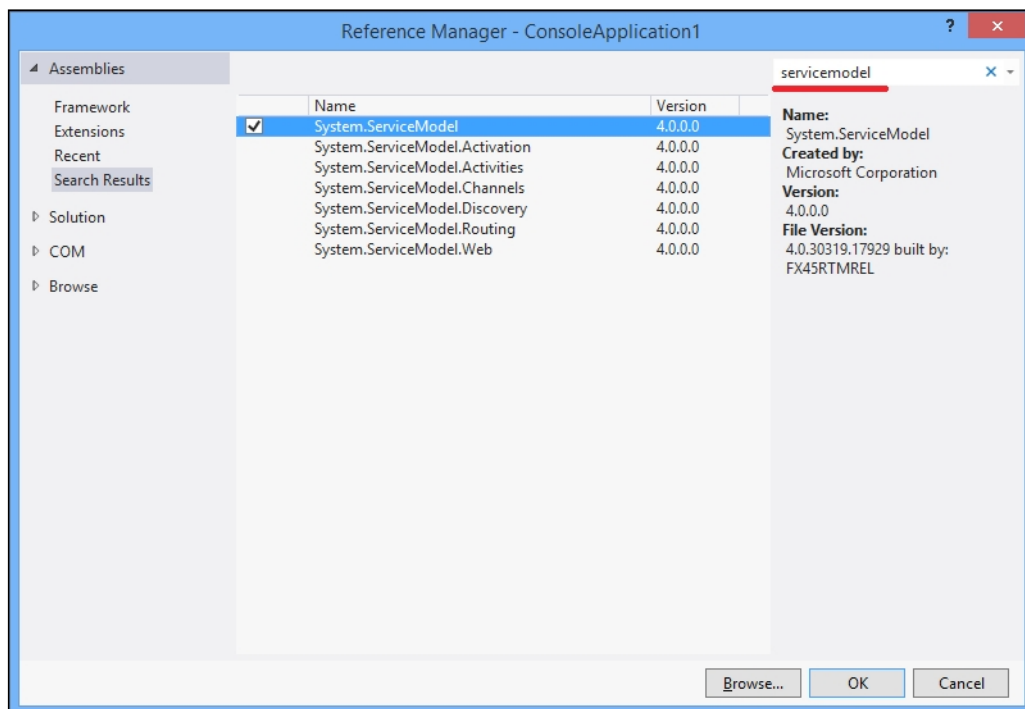
Getting ready

To step through this recipe, you will need Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter9\Recipe4`.

How to do it...

To understand how to work with a WCF service, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add references to the `System.ServiceModel` library. Right-click on the `References` folder in the project and select the **Add reference...** menu option. Add references to the `System.ServiceModel` library. You can use the search function in the reference manager dialog, as shown in the following screenshot:



3. In the `Program.cs` file, add the following using directives:

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.Threading.Tasks;
using static System.Console;
```

4. Add the following code snippet below the `Main` method:

```
const string SERVICE_URL = "http://localhost:1234/HelloWorld";

static async Task RunServiceClient()
{
    var endpoint = new EndpointAddress(SERVICE_URL);
    var channel = ChannelFactory<IHelloWorldServiceClient>
        .CreateChannel(new BasicHttpBinding(), endpoint);

    var greeting = await channel.GreetAsync("Eugene");
    WriteLine(greeting);
}

[ServiceContract(Namespace = "Packt", Name =
"HelloWorldServiceContract")]
public interface IHelloWorldService
{
    [OperationContract]
    string Greet(string name);
}

[ServiceContract(Namespace = "Packt", Name =
"HelloWorldServiceContract")]
public interface IHelloWorldServiceClient
{
    [OperationContract]
    string Greet(string name);

    [OperationContract]
    Task<string> GreetAsync(string name);
}

public class HelloWorldService : IHelloWorldService
{
    public string Greet(string name)
    {
        return $"Greetings, {name}!";
    }
}
```

5. Add the following code snippet inside the Main method:

```
ServiceHost host = null;

try
{
    host = new ServiceHost(typeof (HelloWorldService), new
Uri(SERVICE_URL));
    var metadata =
host.Description.Behaviors.Find<ServiceMetadataBehavior>()
    ?? new ServiceMetadataBehavior();

    metadata.HttpGetEnabled = true;
    metadata.MetadataExporter.PolicyVersion =
PolicyVersion.Policy15;
    host.Description.Behaviors.Add(metadata);

    host.AddServiceEndpoint(ServiceMetadataBehavior.MexContractName,
        MetadataExchangeBindings.CreateMexHttpBinding(), "mex");

    var endpoint = host.AddServiceEndpoint(typeof
(IHelloWorldService),new BasicHttpBinding(), SERVICE_URL);

    host.Faulted += (sender, e) => WriteLine("Error!");

    host.Open();

    WriteLine("Greeting service is running and listening on:");
    WriteLine($"{endpoint.Address} ({endpoint.Binding.Name})");

    var client = RunServiceClient();
    client.GetAwaiter().GetResult();

    WriteLine("Press Enter to exit");
    ReadLine();
}
catch (Exception ex)
{
    WriteLine($"Error in catch block: {ex}");
}
finally
{
    if (null != host)
    {
```

```

        if (host.State == CommunicationState.Faulted)
        {
            host.Abort();
        }
        else
        {
            host.Close();
        }
    }
}

```

6. Run the program.

How it works...

WCF is a framework that allows us to call remote services in different ways. One of them, which was very popular some time ago, was used to call remote services via HTTP using an XML-based protocol called the **Simple Object Access Protocol (SOAP)**. It is quite common when a server application calls another remote service, and this could be done using I/O threads as well.

Visual Studio 2015 has rich support for WCF services; for example, you can add references to such services with the **Add Service Reference** menu option. You could do this with our service as well because we provide service metadata.

To create such a service, we need to use a `ServiceHost` class that will host our service. We describe what service we will be hosting by providing a service implementation type and the base URI by which the service will be addressed. Then, we configure the metadata endpoint and the service endpoint. Finally, we handle the `Faulted` event in case of errors and run the host service.



Be aware that we need to have administrator privileges to run the service, since it uses HTTP bindings, which in turn use `http.sys` and thus require special permissions to be created. You can run Visual Studio under an administrator or run the following command in the elevated command prompt to add the necessary permissions:

```
netsh http add urlacl url=http://+:1234/HelloWorld
user=machine\user
```

To consume this service, we create a client, and here is where the main trick happens. On the server side, we have a service with the usual synchronous method called `Greet`. This method is defined in the service contract, `IHellWorldService`. However, if we want to leverage an asynchronous network I/O, we have to call this method asynchronously. We can do that by creating a new service contract with a matching namespace and service name, where we define both the synchronous and task-based asynchronous methods. In spite of the fact that we do not have an asynchronous method definition on the server side, we follow the naming convention, and the WCF infrastructure understands that we want to create an asynchronous proxy method.

Therefore, when we create an `IHellWorldServiceClient` proxy channel, and WCF correctly routes an asynchronous call to the server-side synchronous method, if you leave the application running, you can open the browser and access the service using its URL, that is, `http://localhost:1234/HellWorld`. A service description will be opened, and you can browse to the XML metadata that allows us to add a service reference from Visual Studio 2012. If you try to generate the reference, you will see slightly more complicated code, but it is autogenerated and easy to use.

10

Parallel Programming Patterns

In this chapter, we will review the common problems that a programmer often faces while trying to implement a parallel workflow. You will learn the following recipes:

- ▶ Implementing Lazy-evaluated shared states
- ▶ Implementing Parallel Pipeline with `BlockingCollection`
- ▶ Implementing Parallel Pipeline with TPL `DataFlow`
- ▶ Implementing Map/Reduce with PLINQ

Introduction

Patterns in programming means a concrete and standard solution to a given problem. Usually, programming patterns are the result of people gathering experience, analyzing the common problems, and providing solutions to these problems.

Since parallel programming has existed for quite a long time, there are many different patterns that are used to program parallel applications. There are even special programming languages to make programming of specific parallel algorithms easier. However, this is where things start to become increasingly complicated. In this chapter, I will provide you with a starting point from where you will be able to study parallel programming further. We will review very basic, yet very useful, patterns that are quite helpful for many common situations in parallel programming.

First, we will be using a **shared-state object** from multiple threads. I would like to emphasize that you should avoid it as much as possible. As we discussed in previous chapters, a shared state is really bad when you write parallel algorithms, but on many occasions, it is inevitable. We will find out how to delay the actual computation of an object until it is needed and how to implement different scenarios to achieve thread safety.

Then, we will show you how to create a structured parallel data flow. We will review a concrete case of a producer/consumer pattern, which is called **Parallel Pipeline**. We are going to implement it by just blocking the collection first, and then we will see how helpful another library from Microsoft is for parallel programming—**TPL DataFlow**.

The last pattern that we will study is the **Map/Reduce** pattern. In the modern world, this name could mean very different things. Some people consider Map/Reduce not as a common approach to any problem, but as a concrete implementation for large, distributed cluster computations. We will find out the meaning behind the name of this pattern and review some examples of how it might work in cases of small parallel applications.

Implementing Lazy-evaluated shared states

This recipe shows how to program a Lazy-evaluated, thread-safe shared state object.

Getting ready

To start this recipe, you will need to run Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter10\Recipe1`.

How to do it...

To implement Lazy-evaluated shared states, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
static async Task ProcessAsynchronously()
{
    var unsafeState = new UnsafeState();
    Task[] tasks = new Task[4];

    for (int i = 0; i < 4; i++)
    {
        tasks[i] = Task.Run(() => Worker(unsafeState));
    }
}
```

```

        await Task.WhenAll(tasks);
        WriteLine(" ----- ");

        var firstState = new DoubleCheckedLocking();
        for (int i = 0; i < 4; i++)
        {
            tasks[i] = Task.Run(() => Worker(firstState));
        }

        await Task.WhenAll(tasks);
        WriteLine(" ----- ");

        var secondState = new BCLDoubleChecked();
        for (int i = 0; i < 4; i++)
        {
            tasks[i] = Task.Run(() => Worker(secondState));
        }

        await Task.WhenAll(tasks);
        WriteLine(" ----- ");

        var lazy = new Lazy<ValueToAccess>(Compute);
        var thirdState = new LazyWrapper(lazy);
        for (int i = 0; i < 4; i++)
        {
            tasks[i] = Task.Run(() => Worker(thirdState));
        }

        await Task.WhenAll(tasks);
        WriteLine(" ----- ");

        var fourthState = new BCLThreadSafeFactory();
        for (int i = 0; i < 4; i++)
        {
            tasks[i] = Task.Run(() => Worker(fourthState));
        }

        await Task.WhenAll(tasks);
        WriteLine(" ----- ");

    }

    static void Worker(IHasValue state)
    {

```

```
        WriteLine($"Worker runs on thread id {CurrentThread.ManagedThreadId}");
        WriteLine($"State value: {state.Value.Text}");
    }

    static ValueToAccess Compute()
    {
        WriteLine("The value is being constructed on a thread " +
            $"id {CurrentThread.ManagedThreadId}");
        Sleep(TimeSpan.FromSeconds(1));
        return new ValueToAccess(
            $"Constructed on thread id {CurrentThread.ManagedThreadId}");
    }

    class ValueToAccess
    {
        private readonly string _text;
        public ValueToAccess(string text)
        {
            _text = text;
        }

        public string Text => _text;
    }

    class UnsafeState : IHasValue
    {
        private ValueToAccess _value;

        public ValueToAccess Value => _value ?? (_value = Compute());
    }

    class DoubleCheckedLocking : IHasValue
    {
        private readonly object _syncRoot = new object();
        private volatile ValueToAccess _value;

        public ValueToAccess Value
        {
            get
            {
                if (_value == null)
                {

```

```

        lock (_syncRoot)
        {
            if (_value == null) _value = Compute();
        }
    }
    return _value;
}
}

class BCLDoubleChecked : IHasValue
{
    private object _syncRoot = new object();
    private ValueToAccess _value;
    private bool _initialized;

    public ValueToAccess Value => LazyInitializer.EnsureInitialized(
        ref _value, ref _initialized, ref _syncRoot, Compute);
}

class BCLThreadSafeFactory : IHasValue
{
    private ValueToAccess _value;

    public ValueToAccess Value => LazyInitializer.
EnsureInitialized(ref _value, Compute);
}

class LazyWrapper : IHasValue
{
    private readonly Lazy<ValueToAccess> _value;

    public LazyWrapper(Lazy<ValueToAccess> value )
    {
        _value = value;
    }

    public ValueToAccess Value => _value.Value;
}

interface IHasValue
{
    ValueToAccess Value { get; }
}

```


4. Add the following code snippet inside the `Main` method:

```
var t = ProcessAsynchronously();  
t.GetAwaiter().GetResult();
```

5. Run the program.

How it works...

The first example shows why it is not safe to use the `UnsafeState` object with multiple accessing threads. We see that the `Construct` method was called several times, and different threads use different values, which is obviously not right. To fix this, we can use a lock when reading the value, and if it is not initialized, create it first. This will work, but using a lock with every read operation is not efficient. To avoid using locks every time, we can use a traditional approach called the **double-checked locking** pattern. We check the value for the first time, and if it is not null, we avoid unnecessary locking and just use the shared object. However, if it was not constructed, we use the lock and then check the value for the second time because it could be initialized between our first check and the lock operation. If it is still not initialized, only then do we compute the value. We can clearly see that this approach works with the second example—there is only one call to the `Construct` method, and the first-called thread defines the shared object state.



Note that if the Lazy-evaluated object implementation is thread-safe, it does not automatically mean that all its properties are thread-safe as well.

If you add, for example, an `int` public property to the `ValueToAccess` object, it will not be thread-safe; you still have to use interlocked constructs or locking to ensure thread safety.

This pattern is very common, and that is why there are several classes in the Base Class Library to help us. First, we can use the `LazyInitializer.EnsureInitialized` method, which implements the double-checked locking pattern inside. However, the most comfortable option is to use the `Lazy<T>` class, which allows us to have thread-safe, Lazy-evaluated, shared state, out of the box. The next two examples show us that they are equivalent to the second one, and the program behaves in the same way. The only difference is that since `LazyInitializer` is a static class, we do not have to create a new instance of a class, as we do in the case of `Lazy<T>`, and therefore, the performance in the first case can be better in some rare scenarios.

The last option is to avoid locking at all if we do not care about the `Construct` method. If it is thread-safe and has no side effects/serious performance impacts, we can just run it several times but use only the first constructed value. The last example shows the described behavior, and we can achieve this result using another `LazyInitializer.EnsureInitialized` method overload.

Implementing Parallel Pipeline with BlockingCollection

This recipe will describe how to implement a specific scenario of a producer/consumer pattern, which is called Parallel Pipeline, using the standard `BlockingCollection` data structure.

Getting ready

To begin this recipe, you will need to run Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter10\Recipe2`.

How to do it...

To understand how to implement Parallel Pipeline using `BlockingCollection`, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:

```
using System;
using System.Collections.Concurrent;
using System.Globalization;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using static System.Console;
using static System.Threading.Thread;
```

3. Add the following code snippet below the `Main` method:

```
private const int CollectionsNumber = 4;
private const int Count = 5;

static void CreateInitialValues(BlockingCollection<int>[]
sourceArrays, CancellationTokenSource cts)
{
    Parallel.For(0, sourceArrays.Length*Count, (j, state) =>
    {
        if (cts.Token.IsCancellationRequested)
        {
            state.Stop();
        }
    })
```

```

        int number = GetRandomNumber(j);
        int k = BlockingCollection<int>.TryAddToAny(sourceArrays,
j);
        if (k >= 0)
        {
            WriteLine(
                $"added {j} to source data on thread " +
                $"id {CurrentThread.ManagedThreadId}");
            Sleep(TimeSpan.FromMilliseconds(number));
        }
    });
    foreach (var arr in sourceArrays)
    {
        arr.CompleteAdding();
    }
}

static int GetRandomNumber(int seed)
{
    return new Random(seed).Next(500);
}

class PipelineWorker<TInput, TOutput>
{
    Func<TInput, TOutput> _processor;
    Action<TInput> _outputProcessor;
    BlockingCollection<TInput>[] _input;
    CancellationToken _token;
    Random _rnd;

    public PipelineWorker(
        BlockingCollection<TInput>[] input,
        Func<TInput, TOutput> processor,
        CancellationToken token,
        string name)
    {
        _input = input;
        Output = new BlockingCollection<TOutput>[_input.Length];
        for (int i = 0; i < Output.Length; i++)
            Output[i] = null == input[i] ? null
                : new BlockingCollection<TOutput>(Count);

        _processor = processor;
        _token = token;
    }
}

```

```

        Name = name;
        _rnd = new Random(DateTime.Now.Millisecond);
    }

    public PipelineWorker(
        BlockingCollection<TInput>[] input,
        Action<TInput> renderer,
        CancellationToken token,
        string name)
    {
        _input = input;
        _outputProcessor = renderer;
        _token = token;
        Name = name;
        Output = null;
        _rnd = new Random(DateTime.Now.Millisecond);
    }

    public BlockingCollection<TOutput>[] Output { get; private set; }

    public string Name { get; private set; }

    public void Run()
    {
        WriteLine($"{Name} is running");
        while (!_input.All(bc => bc.IsCompleted) &&
            !_token.IsCancellationRequested)
        {
            TInput receivedItem;
            int i = BlockingCollection<TInput>.TryTakeFromAny(
                _input, out receivedItem, 50, _token);
            if (i >= 0)
            {
                if (Output != null)
                {
                    TOutput outputItem = _processor(receivedItem);
                    BlockingCollection<TOutput>.AddToAny(
                        Output, outputItem);
                    WriteLine($"{Name} sent {outputItem} to next, on " +
                        $"thread id {CurrentThread.ManagedThreadId}");
                    Sleep(TimeSpan.FromMilliseconds(_rnd.Next(200)));
                }
                else

```



```
        {
            _outputProcessor(receivedItem);
        }
    }
    else
    {
        Sleep(TimeSpan.FromMilliseconds(50));
    }
}
if (Output != null)
{
    foreach (var bc in Output) bc.CompleteAdding();
}
}
```

4. Add the following code snippet inside the Main method:

```
var cts = new CancellationTokenSource();

Task.Run(() =>
{
    if (ReadKey().KeyChar == 'c') cts.Cancel();
},
cts.Token);

var sourceArrays = new BlockingCollection<int>[CollectionsNumber];

for (int i = 0; i < sourceArrays.Length; i++)
{
    sourceArrays[i] = new BlockingCollection<int>(Count);
}

var convertToDecimal = new PipelineWorker<int, decimal>
(
    sourceArrays,
    n => Convert.ToDecimal(n*100),
    cts.Token,
    "Decimal Converter"
);

var stringifyNumber = new PipelineWorker<decimal, string>
(
    convertToDecimal.Output,
```

```

        s => $"--{s.ToString("C", CultureInfo.GetCultureInfo("en-us"))}--",
        cts.Token,
        "String Formatter"
    );

    var outputResultToConsole = new PipelineWorker<string, string>
    (
        stringifyNumber.Output,
        s => WriteLine($"The final result is {s} on thread " +
            $"id {CurrentThread.ManagedThreadId}"),
        cts.Token,
        "Console Output"
    );

    try
    {
        Parallel.Invoke(
            () =>
                CreateInitialValues(sourceArrays, cts),
            () => convertToDecimal.Run(),
            () => stringifyNumber.Run(),
            () => outputResultToConsole.Run()
        );
    }
    catch (AggregateException ae)
    {
        foreach (var ex in ae.InnerExceptions)
            WriteLine(ex.Message + ex.StackTrace);
    }

    if (cts.Token.IsCancellationRequested)
    {
        WriteLine("Operation has been canceled! Press ENTER to exit.");
    }
    else
    {
        WriteLine("Press ENTER to exit.");
    }
    ReadLine();

```

5. Run the program.

How it works...

In the preceding example, we implement one of the most common parallel programming scenarios. Imagine that we have some data that has to pass through several computation stages, which takes a significant amount of time. The latter computation requires the results of the former, so we cannot run them in parallel.

If we had only one item to process, there would not be many possibilities to enhance the performance. However, if we run many items through the same set of computation stages, we can use a Parallel Pipeline technique. This means that we do not have to wait until all items pass through the first computation stage to go to the next one. It is enough to have just one item that finishes the stage; we move it to the next stage, and meanwhile, the next item is being by the previous stage, and so on. As a result, we almost have parallel processing shifted by the time required for the first item to pass through the first computation stage.

Here, we use four collections for each processing stage, illustrating that we can process every stage in parallel as well. The first step that we do is to provide the possibility to cancel the whole process by pressing the C key. We create a cancelation token and run a separate task to monitor the C key. Then, we define our pipeline. It consists of three main stages. The first stage is where we put the initial numbers on the first four collections that serve as the item source to the latter pipeline. This code is inside the `Parallel.For` loop of the `CreateInitialValues` method, which in turn is inside the `Parallel.Invoke` statement, as we run all the stages in parallel; the initial stage runs in parallel as well.

The next stage is defining our pipeline elements. The logic is defined inside the `PipelineWorker` class. We initialize the worker with the input collection, provide a transformation function, and then run the worker in parallel with the other workers. This way, we define two workers, or filters, because they filter the initial sequence. One of them turns an integer into a decimal value, and the second one turns a decimal to a string. Finally, the last worker just prints every incoming string to the console. In all the places, we provide a running thread ID to see how everything works. Besides this, we added artificial delays, so the item's processing will be more natural, as we really use heavy computations.

As a result, we see the exact expected behavior. First, some items are created on the initial collections. Then, we see that the first filter starts to process them, and as they are being processed, the second filter starts to work. Finally, the item goes to the last worker that prints it to the console.

Implementing Parallel Pipeline with TPL DataFlow

This recipe shows how to implement a Parallel Pipeline pattern with the help of the TPL DataFlow library.

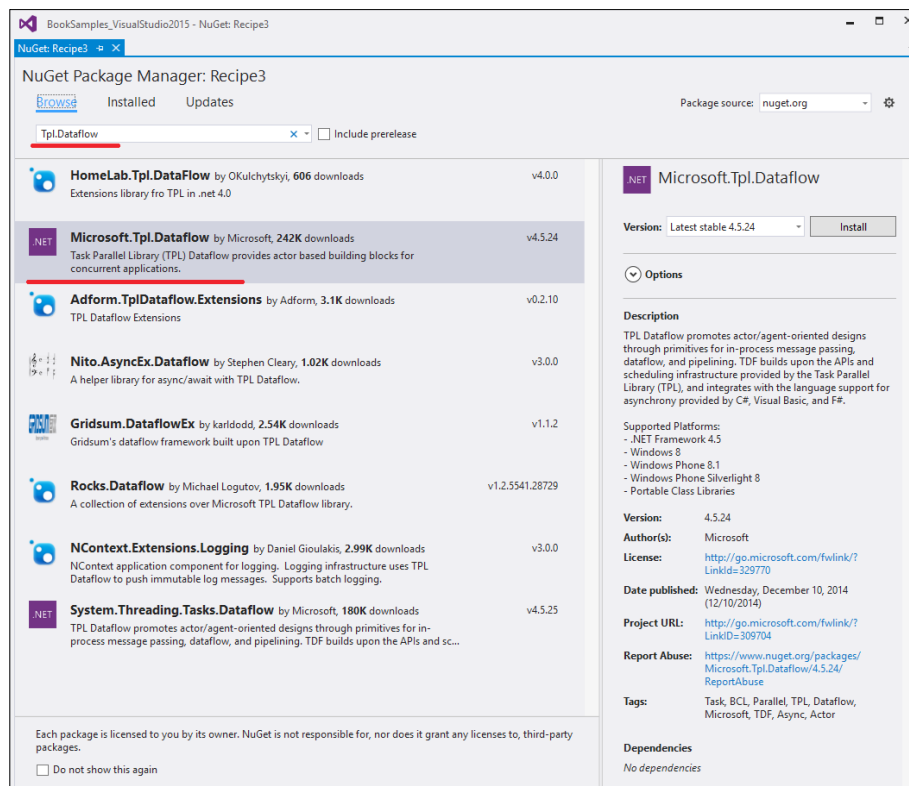
Getting ready

To start this recipe, you will need to run Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter10\Recipe3`.

How to do it...

To understand how to implement Parallel Pipeline with TPL DataFlow, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Add references to the **Microsoft TPL DataFlow** NuGet package. Follow these steps to do so:
 1. Right-click on the **References** folder in the project and select the **Manage NuGet Packages...** menu option.
 2. Now, add your preferred references to the **Microsoft TPL DataFlow** NuGet package. You can use the search option in the **Manage NuGet Packages** dialog as follows:



3. In the `Program.cs` file, add the following using directives:

```
using System;
using System.Globalization;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using static System.Console;
using static System.Threading.Thread;
```

4. Add the following code snippet below the `Main` method:

```
async static Task ProcessAsynchronously()
{
    var cts = new CancellationTokenSource();
    Random _rnd = new Random(DateTime.Now.Millisecond);

    Task.Run(() =>
    {
        if (ReadKey().KeyChar == 'c')
            cts.Cancel();
    }, cts.Token);

    var inputBlock = new BufferBlock<int>(
        new DataflowBlockOptions { BoundedCapacity = 5,
        CancellationToken = cts.Token });

    var convertToDecimalBlock = new TransformBlock<int, decimal>(
        n =>
        {
            decimal result = Convert.ToDecimal(n * 100);
            WriteLine($"Decimal Converter sent {result} to the next
stage on " +
                $"thread id {CurrentThread.ManagedThreadId}");
            Sleep(TimeSpan.FromMilliseconds(_rnd.Next(200)));
            return result;
        },
        new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism =
4, CancellationToken = cts.Token });

    var stringifyBlock = new TransformBlock<decimal, string>(
        n =>
        {
            string result = $"--{n.ToString("C", CultureInfo.
GetCultureInfo("en-us"))}--";
```

```

        WriteLine($"String Formatter sent {result} to the next stage
on thread id {CurrentThread.ManagedThreadId}");
        Sleep(TimeSpan.FromMilliseconds(_rnd.Next(200)));
        return result;
    }
    , new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism =
4, CancellationToken = cts.Token });

    var outputBlock = new ActionBlock<string>(
        s =>
        {
            WriteLine($"The final result is {s} on thread id
{CurrentThread.ManagedThreadId}");
        }
        , new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism =
4, CancellationToken = cts.Token });

    inputBlock.LinkTo(convertToDecimalBlock, new DataflowLinkOptions
{ PropagateCompletion = true });
    convertToDecimalBlock.LinkTo(stringifyBlock, new
DataflowLinkOptions { PropagateCompletion = true });
    stringifyBlock.LinkTo(outputBlock, new DataflowLinkOptions {
PropagateCompletion = true });

    try
    {
        Parallel.For(0, 20, new ParallelOptions {
MaxDegreeOfParallelism = 4, CancellationToken = cts.Token }
        , i =>
        {
            WriteLine($"added {i} to source data on thread id
{CurrentThread.ManagedThreadId}");
            inputBlock.SendAsync(i).GetAwaiter().GetResult();
        });
        inputBlock.Complete();
        await outputBlock.Completion;
        WriteLine("Press ENTER to exit.");
    }
    catch (OperationCanceledException)
    {
        WriteLine("Operation has been canceled! Press ENTER to
exit.");
    }

    ReadLine();
}

```

5. Add the following code snippet inside the `Main` method:

```
var t = ProcessAsynchronously();  
t.GetAwaiter().GetResult();
```

6. Run the program.

How it works...

In the previous recipe, we implemented a Parallel Pipeline pattern to process items through sequential stages. It is quite a common problem, and one of the proposed ways to program such algorithms is using a TPL DataFlow library from Microsoft. It is distributed via NuGet and is easy to install and use in your application.

The TPL DataFlow library contains different types of blocks that can be connected with each other in different ways and form complicated processes that can be partially parallel and sequential where needed. To see some of the available infrastructure, let's implement the previous scenario with the help of the TPL DataFlow library.

First, we define the different blocks that will be processing our data. Note that these blocks have different options that can be specified during their construction; they can be very important. For example, we pass the cancelation token into every block we define, and when we signal the cancelation, all of them stop working.

We start our process with `BufferBlock`, we bound its capacity to 5 items maximum. This block holds items to pass them to the next blocks in the flow. We restrict it to the five-item capacity, specifying the `BoundedCapacity` option value. This means that when there will be five items in this block, it will stop accepting new items until one of the existing items passes to the next blocks.

The next block type is `TransformBlock`. This block is intended for a data transformation step. Here, we define two transformation blocks; one of them creates decimals from integers, and the second one creates a string from a decimal value. We can use the `MaxDegreeOfParallelism` option for this block, specifying the maximum simultaneous worker threads.

The last block is of the `ActionBlock` type. This block will run a specified action on every incoming item. We use this block to print our items to the console.

Now, we link these blocks together with the help of the `LinkTo` methods. Here, we have an easy sequential data flow, but it is possible to create schemes that are more complicated. Here, we also provide `DataflowLinkOptions` with the `PropagateCompletion` property set to `true`. This means that when the step is complete, it will automatically propagate its results and exceptions to the next stage. Then, we start adding items to the buffer block in parallel, calling the block's `Complete` method, when we finish adding new items. Then, we wait for the last block to get completed. In the case of a cancelation, we handle `OperationCancelledException` and cancel the whole process.

Implementing Map/Reduce with PLINQ

This recipe will describe how to implement the Map/Reduce pattern while using PLINQ.

Getting ready

To begin this recipe, you will need to run Visual Studio 2015. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter10\Recipe4`.

How to do it...

To understand how to implement Map/Reduce with PLINQ, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. In the `Program.cs` file, add the following `using` directives:


```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

using Newtonsoft.Json;

using static System.Console;
```
3. Add references to the `Newtonsoft.Json` NuGet package and the `System.Net.Http` assembly.
4. Add the following code snippet below the `Main` method:

```
static char[] delimiters = { ' ', ',', ';', ':', '\"', '.' };

async static Task<string> ProcessBookAsync(
    string bookContent, string title, HashSet<string> stopwords)
{
    using (var reader = new StringReader(bookContent))
    {
        var query = reader.EnumerateLines()
            .AsParallel()
            .SelectMany(line => line.Split(delimiters))
            .MapReduce(
```



```
        word => new[] { word.ToLower() },
        key => key,
        g => new[] { new { Word = g.Key, Count = g.Count() } }
    } }

    )
    .ToList();

var words = query
    .Where(element =>
        !string.IsNullOrEmpty(element.Word)
        && !stopwords.Contains(element.Word))
    .OrderByDescending(element => element.Count);

var sb = new StringBuilder();

sb.AppendLine($"'{title}' book stats");
sb.AppendLine("Top ten words used in this book: ");
foreach (var w in words.Take(10))
{
    sb.AppendLine($"Word: '{w.Word}', times used: '{w.Count}'");
}

sb.AppendLine($"Unique Words used: {query.Count()}");

return sb.ToString();
}
}

async static Task<string> DownloadBookAsync(string bookUrl)
{
    using (var client = new HttpClient())
    {
        return await client.GetStringAsync(bookUrl);
    }
}

async static Task<HashSet<string>> DownloadStopWordsAsync()
{
    string url =
        "https://raw.githubusercontent.com/6/stopwords/master/stopwords-all.json";

    using (var client = new HttpClient())
```

```

    {
        try
        {
            var content = await client.GetStringAsync(url);
            var words =
                JsonConvert.DeserializeObject<
                    Dictionary<string, string[]>>(content);
            return new HashSet<string>(words["en"]);
        }
        catch
        {
            return new HashSet<string>();
        }
    }
}

```

5. Add the following code snippet inside the Main method:

```

var booksList = new Dictionary<string, string>()
{
    ["Moby Dick; Or, The Whale by Herman Melville"]
    = "http://www.gutenberg.org/cache/epub/2701/pg2701.txt",

    ["The Adventures of Tom Sawyer by Mark Twain"]
    = "http://www.gutenberg.org/cache/epub/74/pg74.txt",

    ["Treasure Island by Robert Louis Stevenson"]
    = "http://www.gutenberg.org/cache/epub/120/pg120.txt",

    ["The Picture of Dorian Gray by Oscar Wilde"]
    = "http://www.gutenberg.org/cache/epub/174/pg174.txt"
};

HashSet<string> stopwords = DownloadStopWordsAsync().GetAwaiter().
    GetResult();

var output = new StringBuilder();

Parallel.ForEach(booksList.Keys, key =>
{
    var bookContent = DownloadBookAsync(booksList[key])
        .GetAwaiter().GetResult();

```

```
        string result = ProcessBookAsync(bookContent, key, stopwords)
            .GetAwaiter().GetResult();

        output.Append(result);
        output.AppendLine();
    });

    Write(output.ToString());
    ReadLine();
```

6. Add the following code snippet after the Program class definition:

```
static class Extensions
{
    public static ParallelQuery<TResult> MapReduce<TSource, TMapped,
    TKey, TResult>(
        this ParallelQuery<TSource> source,
        Func<TSource, IEnumerable<TMapped>> map,
        Func<TMapped, TKey> keySelector,
        Func<IGrouping<TKey, TMapped>, IEnumerable<TResult>> reduce)
    {
        return source.SelectMany(map)
            .GroupBy(keySelector)
            .SelectMany(reduce);
    }

    public static IEnumerable<string> EnumLines(this StringReader
    reader)
    {
        while (true)
        {
            string line = reader.ReadLine();
            if (null == line) yield break;

            yield return line;
        }
    }
}
```

7. Run the program.

How it works...

The `Map/Reduce` functions are another important parallel programming pattern. They are suitable for a small program and large multiserver computations. The meaning of this pattern is that you have two special functions to apply to your data. The first of them is the `Map` function. It takes a set of initial data in a key/value list form and produces another key/value sequence, transforming the data to a comfortable format for further processing. Then, we use another function, called `Reduce`. The `Reduce` function takes the result of the `Map` function and transforms it to the smallest possible set of data that we actually need. To understand how this algorithm works, let's look through the preceding recipe.

Here, we are going to analyze four classic books' text. We are going to download the books from the project Gutenberg's site (www.gutenberg.org), which can ask for a captcha if you issue many network requests and thus break the program logic of this sample. If you see HTML elements in the program's output, open one of the book URLs in the browser and complete the captcha. The next thing to do is to load a list of English words that we are going to skip when analyzing the text. In this sample, we try to load a JSON-encoded word list from GitHub, and in case of failure, we just get an empty list.

Now, let's pay attention to our `Map/Reduce` implementation as a PLINQ extension method in the `PLINQExtensions` class. We use `SelectMany` to transform the initial sequence to the sequence we need by applying the `Map` function. This function produces several new elements from one sequence element. Then, we choose how we group the new sequence with the `keySelector` function, and we use `GroupBy` with this key to produce an intermediate key/value sequence. The last thing we do is apply `Reduce` to the resulting grouped sequence to get the result.

Then, we run all our books processing in parallel. Each processing worker thread outputs the resulting information into a string, and after all workers are complete, we print this information to the console. We do this to avoid concurrent console output, when each worker text overlaps and makes the resulting information unreadable. In each worker process, we split the book text into a text lines sequence, chop each line into word sequences, and apply our `MapReduce` function to it. We use the `Map` function to transform each word into lowercase and use it as the grouping key. Then, we define the `Reduce` function as a transformation of the grouping element into a key value pair, which has the `Word` element that contains one unique word found in the text and the `Count` element, which has information about how many times this word has been used. The final step is our query materialization with the `ToList` method call, since we need to process this query twice. Then, we use our list of stop words to remove common words from our statistics and create a string result with the book's title, top 10 words used in the book, and a unique word's frequency in the book.

11

There's More

In this chapter, we will look through a new programming paradigm in the Windows 10 operating system. Also, you will learn how to run .NET programs on OS X and Linux. You will learn the following recipes in this chapter:

- ▶ Using a timer in a Universal Windows Platform application
- ▶ Using WinRT from usual applications
- ▶ Using `BackgroundTask` in Universal Windows Platform applications
- ▶ Running a .NET Core application on OS X
- ▶ Running a .NET Core application on Ubuntu Linux

Introduction

Microsoft released the first public beta build of Windows 8 at the Build conference on September 13, 2011. The new OS tried to address almost every problem that Windows had by introducing features such as a responsive UI suitable for tablet devices with touch, lower power consumption, a new application model, new asynchronous APIs, and tighter security.

The core of Windows API improvements was a new multiplatform component system, **WinRT**, which is a logical development of COM. With WinRT, a programmer can use native C++ code, C# and .NET, and even JavaScript and HTML to develop applications. Another change is the introduction of a centralized application store, which did not exist on the Windows platform before.

Being a new application platform, Windows 8 had backward compatibility and allowed us to run the usual Windows applications. This led to a situation where there were two major classes of applications: the Windows Store applications, where new programs are distributed via the Windows Store, and the usual classic applications that had not changed since the previous version of Windows.

However, Windows 8 was only the first step toward the new application model. Microsoft got a lot of feedback from the users, and it became clear that Windows Store applications were too different from what people were used to. Besides that, there was a separate smartphone OS, Windows 8 Phone, that had a different application store and a slightly different set of APIs. This made an application developer create two separate applications for desktop and smartphone platforms.

To improve the situation, the new Windows 10 OS was introduced as a unified platform for all Windows-powered devices. There is a single application store that supports every device family, and now, it is possible to create an application that works on phones, tablets, and desktops. Thus, Windows Store applications are now called Universal Windows Platform applications (UWP apps). This, of course, means a lot of limitations for your application—it should not use any platform-specific APIs, and as a programmer, you have to comply with specific rules. The program has to respond in a limited time to start up or to finish, keeping the whole operating system and other applications responsive. To save the battery, your applications are no longer running in the background by default; instead of that, they get suspended and actually stop executing.

New Windows APIs are asynchronous, and you can only use whitelisted API functions in your application. For example, you are not allowed to create a new `Thread` class instance anymore. You have to use a system-managed thread pool instead. A lot of usual APIs cannot be used anymore, and you have to study new ways to achieve the same goals as before.

But this is not all. Microsoft began to understand that supporting operating systems other than Windows is also important. And now, you can write cross-platform applications using a new subset of .NET that is called .NET Core. Its source can be found on GitHub, and it is supported on platforms such as OS X and Linux. You can use any text editor, but I would suggest you take a look at Visual Studio Code—a new lightweight, cross-platform code editor, which runs on OS X and Linux and understands the C# syntax well.

In this chapter, we will see how a Universal Windows Platform application is different from the usual Windows application and how we can use some of the WinRT benefits from the usual applications. We will also go through a simplified scenario of a Universal Windows Platform application with background notifications. You will also learn to run a .NET program on OS X and Linux.

Using a timer in a Universal Windows Platform application

This recipe shows you how to use a simple timer in Universal Windows Platform applications.

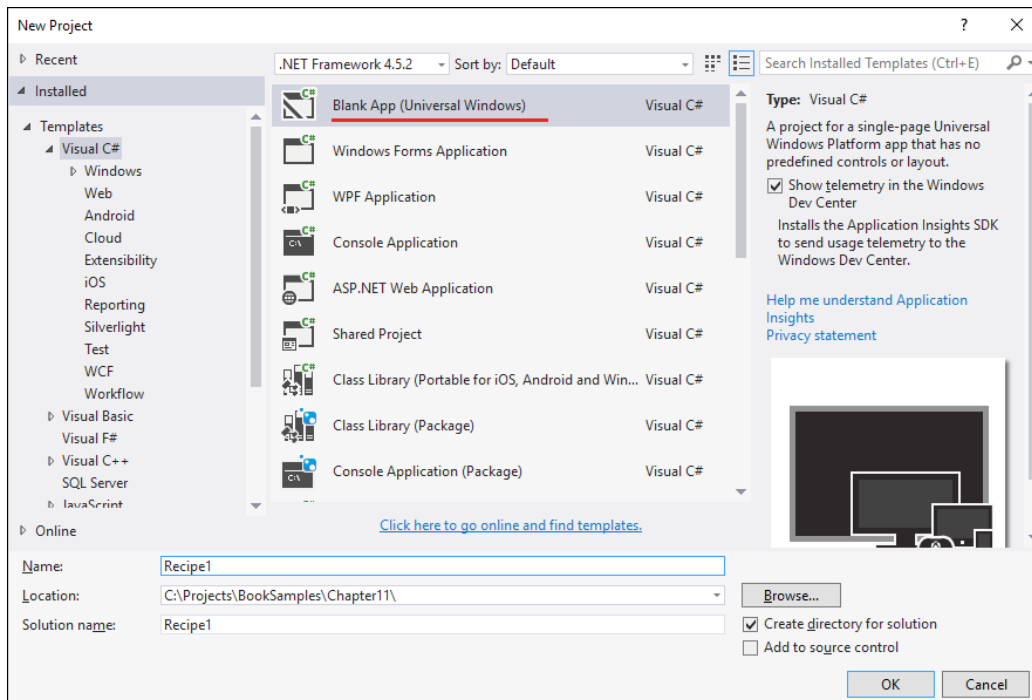
Getting ready

To go through this recipe, you will need Visual Studio 2015 and the Windows 10 operating system. No other prerequisites are required. The source code for this recipe can be found at `BookSamples\Chapter11\Recipe1`.

How to do it...

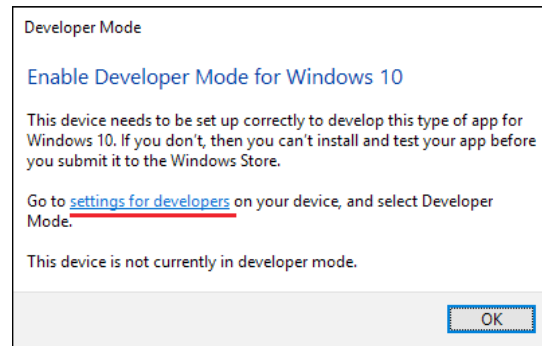
To understand how to use a timer in a Windows Store application, perform the following steps:

1. Start Visual Studio 2015. Create a new C# **Blank App (Universal Windows)** project in the `Windows\Universal` folder.

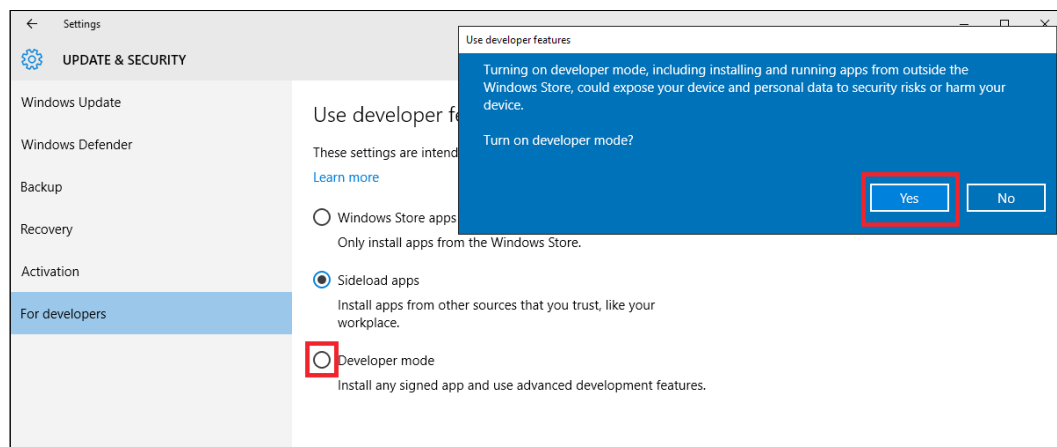


There's More

2. If you are asked to enable developer mode for Windows 10, you have to enable it in the control panel.



3. Then, confirm that you are sure you want to turn on developer mode.



4. In the MainPage.xaml file, add the Name attribute to the Grid element:

```
<Grid Name="Grid" Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
```
5. In the MainPage.xaml.cs file, add the following using directives:

```
using System;  
using Windows.UI.Xaml;  
using Windows.UI.Xaml.Controls;  
using Windows.UI.Xaml.Navigation;
```
6. Add the following code snippet above the MainPage constructor definition:

```
private readonly DispatcherTimer _timer;  
private int _ticks;
```

7. Replace the `MainPage()` constructor with the following code snippet:

```
public MainPage()
{
    InitializeComponent();
    _timer = new DispatcherTimer();
    _ticks = 0;
}
```

8. Add the `OnNavigatedTo()` method under the `MainPage` constructor definition:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
}
```

9. Add the following code snippet inside the `OnNavigatedTo` method:

```
base.OnNavigatedTo(e);
Grid.Children.Clear();
var commonPanel = new StackPanel
{
    Orientation = Orientation.Vertical,
    HorizontalAlignment = HorizontalAlignment.Center
};

var buttonPanel = new StackPanel
{
    Orientation = Orientation.Horizontal,
    HorizontalAlignment = HorizontalAlignment.Center
};

var textBlock = new TextBlock
{
    Text = "Sample timer application",
    FontSize = 32,
    HorizontalAlignment = HorizontalAlignment.Center,
    Margin = new Thickness(40)
};

var timerTextBlock = new TextBlock
{
    Text = "0",
    FontSize = 32,
    HorizontalAlignment = HorizontalAlignment.Center,
    Margin = new Thickness(40)
};
```

```

var timerStateTextBlock = new TextBlock
{
    Text = "Timer is enabled",
    FontSize = 32,
    HorizontalAlignment = HorizontalAlignment.Center,
    Margin = new Thickness(40)
};

var startButton = new Button { Content = "Start",
    FontSize = 32};
var stopButton = new Button { Content = "Stop",
    FontSize = 32};

buttonPanel.Children.Add(startButton);
buttonPanel.Children.Add(stopButton);

commonPanel.Children.Add(textBlock);
commonPanel.Children.Add(timerTextBlock);
commonPanel.Children.Add(timerStateTextBlock);
commonPanel.Children.Add(buttonPanel);

_timer.Interval = TimeSpan.FromSeconds(1);
_timer.Tick += (sender, EventArgs) =>
{
    timerTextBlock.Text = _ticks.ToString(); _ticks++;
};
_timer.Start();

startButton.Click += (sender, EventArgs) =>
{
    timerTextBlock.Text = "0";
    _timer.Start();
    _ticks = 1;
    timerStateTextBlock.Text = "Timer is enabled";
};

stopButton.Click += (sender, EventArgs) =>
{
    _timer.Stop();
    timerStateTextBlock.Text = "Timer is disabled";
};

Grid.Children.Add(commonPanel);

```

10. Right-click on the project in Visual Studio **Solution Explorer** and choose **Deploy**.
11. Run the program.

How it works...

When the program runs, it creates an instance of a `MainPage` class. Here, we instantiate `DispatcherTimer` in the constructor and initialize the `ticks` counter to 0. Then, in the `OnNavigatedTo` event handler, we create our UI controls and bind the start and stop buttons to the corresponding lambda expressions, which contain the `start` and `stop` logics.

As you can see, the `timer` event handler works directly with the UI controls. This is okay because `DispatcherTimer` is implemented in such a way that the handlers of the `Tick` event of `timer` are run by the UI thread. However, if you run the program and then switch to something else and then switch to the program after a couple of minutes, you may notice that the seconds counter is far behind the real amount of time that passed. This happens because Universal Windows Platform applications have completely different life cycles.



Be aware that Universal Windows Platform applications behave much like the applications on smartphone and tablet platforms. Instead of running in the background, they become suspended after some time, and this means that they are actually frozen until the user switches back to them. You have a limited time to save the current application state before it becomes suspended, and you are able to restore the state when the applications run again.

While this behavior could save power and CPU resources, it creates significant difficulties for program applications that are supposed to do some processing in the background. Windows 10 has a set of special APIs to program such applications. We will go through such a scenario later in this chapter.

Using WinRT from usual applications

This recipe shows you how to create a console application that will be able to use the WinRT API.

Getting ready

To go through this recipe, you will need Visual Studio 2015 and the Windows 10 operating system. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter11\Recipe2`.

How to do it...

To understand how to use WinRT from usual applications, perform the following steps:

1. Start Visual Studio 2015. Create a new C# console application project.
2. Right-click on the created project in Visual Studio **Solution Explorer** and select the **Unload Project...** menu option.
3. Right-click on the unloaded project and select the **Edit ProjectName.csproj** menu option.
4. Add the following XML code below the `<TargetFrameworkVersion>` element:
`<TargetPlatformVersion>10.0</TargetPlatformVersion>`
5. Save the `.csproj` file, right-click on the unloaded project in Visual Studio **Solution Explorer**, and select the **Reload Project** menu option.
6. Right-click on the project and select **Add Reference** from the **Core** library under **Windows**. Then, click on the **Browse** button.
7. Navigate to `C:\Program Files (x86)\Windows Kits\10\UnionMetadata` and click on `Windows.winmd`.
8. Navigate to `C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5` and click on the `System.Runtime.WindowsRuntime.dll` file.
9. In the `Program.cs` file, add the following using directives:
`using System;`
`using System.IO;`
`using System.Threading.Tasks;`
`using Windows.Storage;`

10. Add the following code snippet below the `Main` method:

```
async static Task AsynchronousProcessing()
{
    StorageFolder folder = KnownFolders.DocumentsLibrary;

    if (await folder.DoesFileExistAsync("test.txt"))
    {
        var fileToDelete = await folder.GetFilesAsync(
            "test.txt");
        await fileToDelete.DeleteAsync(
            StorageDeleteOption.PermanentDelete);
    }

    var file = await folder.CreateFileAsync("test.txt",
        CreationCollisionOption.ReplaceExisting);
}
```

```

        using (var stream = await file.OpenAsync(FileAccessMode.
ReadWrite))
        using (var writer = new StreamWriter(stream.AsStreamForWrite()))
        {
            await writer.WriteLineAsync("Test content");
            await writer.FlushAsync();
        }

        using (var stream = await file.OpenAsync(FileAccessMode.Read))
        using (var reader = new StreamReader(stream.AsStreamForRead()))
        {
            string content = await reader.ReadToEndAsync();
            Console.WriteLine(content);
        }

        Console.WriteLine("Enumerating Folder Structure:");

        var itemsList = await folder.GetItemsAsync();
        foreach (var item in itemsList)
        {
            if (item is StorageFolder)
            {
                Console.WriteLine("{0} folder", item.Name);
            }
            else
            {
                Console.WriteLine(item.Name);
            }
        }
    }
}

```

11. Add the following code snippet to the Main method:

```

var t = AsynchronousProcessing();
t.GetAwaiter().GetResult();
Console.WriteLine();
Console.WriteLine("Press ENTER to continue");
Console.ReadLine();

```

12. Add the following code snippet below the Program class definition:

```

static class Extensions
{
    public static async Task<bool> DoesFileExistAsync(this
        StorageFolder folder, string fileName)
    {
        try
        {

```

```
        await folder.GetFilesAsync(fileName);  
        return true;  
    }  
    catch (FileNotFoundException)  
    {  
        return false;  
    }  
}
```

13. Run the program.

How it works...

Here, we used quite a tricky way to consume the WinRT API from a common .NET console application. Unfortunately, not all available APIs will work in this scenario, but still, it could be useful to work with movement sensors, GPS location services, and so on.

To reference WinRT in Visual Studio, we manually edit the `.csproj` file, specifying the target platform for the application as Windows 10. Then, we manually reference `Windows.winmd` to get access to Windows 10 APIs and `System.Runtime.WindowsRuntime.dll` to leverage the `GetAwaiter` extension method implementation for WinRT asynchronous operations. This allows us to use `await` on WinRT APIs directly. There is a backward conversion as well. When we create a WinRT library, we have to expose the WinRT native `IAsyncOperation` interfaces family for asynchronous operations, so they could be consumed from JavaScript and C++ in a language-agnostic manner.

File operations in WinRT are quite self-descriptive; here, we have asynchronous file create and delete operations. Still, file operations in WinRT contain security restrictions, encouraging you to use special Windows folders for your application and not allowing you to work with just any file path on your disk drive.

Using BackgroundTask in Universal Windows Platform applications

This recipe walks you through the process of creating a background task in a Universal Windows Platform application, which updates the application's live tile on a desktop.

Getting ready

To go through this recipe, you will need Visual Studio 2015 and the Windows 10 operating system. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter11\Recipe3`.

How to do it...

To understand how to use `BackgroundTask` in Universal Windows Platform applications, perform the following steps:

1. Start Visual Studio 2015. Create a new C# **Blank App (Universal Windows)** project under `Windows\Universal` folder. If you need to enable the Windows 10 developer mode, refer to the *Using a timer in a Windows Store application* recipe for detailed instructions.
2. Open the `Package.appxmanifest` file. In the **Declarations** tab, add **Background Tasks** to **Supported Declarations**. Under **Properties**, check the supported properties **System event** and **Timer** and set the name of **Entry point** to `YourNamespace.TileSchedulerTask`. `YourNamespace` should be the namespace of your application.

The screenshot shows the 'Declarations' tab in the Visual Studio Package.appxmanifest editor. The interface is divided into several sections:

- Available Declarations:** A dropdown menu shows 'Background Tasks' selected, with an 'Add' button next to it.
- Supported Declarations:** A list box shows 'Background Tasks' selected, with a 'Remove' button next to it.
- Description:** Text explaining that this declaration enables the app to specify the class name of an in-proc server DLL that runs the app code in the background in response to external trigger events. It also mentions that multiple instances of this declaration are allowed in each app, with a link to 'More information'.
- Properties:** A section titled 'Supported task types' containing a list of checkboxes:
 - ☐ Audio
 - ☐ Bluetooth
 - ☐ Bluetooth device connection
 - ☐ Chat message notification
 - ☐ Control channel
 - ☐ Device use trigger
 - ☐ General
 - ☐ Location
 - ☐ Media processing
 - ☐ Phone call
 - ☐ Push notification
 - ☒ System event
 - ☒ Timer
 - ☐ VPN client
- App settings:** A section with input fields for:
 - Executable: (empty)
 - Entry point: `Recipe3.TileSchedulerTask`
 - Start page: (empty)
 - Resource group: (empty)

3. In the MainPage.xaml file, insert the following XAML code into the Grid element:

```
<StackPanel Margin="50">
    <TextBlock Name="Clock"
        Text="HH:mm"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Style="{StaticResource HeaderTextBlockStyle}"/>
</StackPanel>
```

4. In the MainPage.xaml.cs file, add the following using directives:

```
using System;
using System.Diagnostics;
using System.Globalization;
using System.Linq;
using System.Xml.Linq;
using Windows.ApplicationModel.Background;
using Windows.Data.Xml.Dom;
using Windows.System.UserProfile;
using Windows.UI.Notifications;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
```

5. Add the following code snippet above the MainPage constructor definition:

```
private const string TASK_NAME_USERPRESENT =
    "TileSchedulerTask_UserPresent";
private const string TASK_NAME_TIMER =
    "TileSchedulerTask_Timer";
```

```
private readonly CultureInfo _cultureInfo;
private readonly DispatcherTimer _timer;
```

6. Replace the MainPage constructor with the following code snippet:

```
public MainPage()
{
    InitializeComponent();

    string language = GlobalizationPreferences.Languages.First();
    _cultureInfo = new CultureInfo(language);

    _timer = new DispatcherTimer();
    _timer.Interval = TimeSpan.FromSeconds(1);
    _timer.Tick += (sender, e) => UpdateClockText();
}
```

7. Add the following code snippet above the `OnNavigatedTo` method:

```
private void UpdateClockText()
{
    Clock.Text = DateTime.Now.ToString(
        _cultureInfo.DateTimeFormat.FullDateTimePattern);
}

private static async void CreateClockTask()
{
    BackgroundAccessStatus result = await
        BackgroundExecutionManager.RequestAccessAsync();
    if (result == BackgroundAccessStatus.
        AllowedMayUseActiveRealTimeConnectivity ||
        result == BackgroundAccessStatus.
            AllowedWithAlwaysOnRealTimeConnectivity)
    {
        TileSchedulerTask.CreateSchedule();

        EnsureUserPresentTask();
        EnsureTimerTask();
    }
}

private static void EnsureUserPresentTask()
{
    foreach (var task in BackgroundTaskRegistration.AllTasks)
        if (task.Value.Name == TASK_NAME_USERPRESENT)
            return;

    var builder = new BackgroundTaskBuilder();
    builder.Name = TASK_NAME_USERPRESENT;
    builder.TaskEntryPoint =
        (typeof(TileSchedulerTask)).FullName;
    builder.SetTrigger(new SystemTrigger(
        SystemTriggerType.UserPresent, false));
    builder.Register();
}

private static void EnsureTimerTask()
{
    foreach (var task in BackgroundTaskRegistration.AllTasks)
        if (task.Value.Name == TASK_NAME_TIMER)
            return;
}
```

```
var builder = new BackgroundTaskBuilder();
builder.Name = TASK_NAME_TIMER;
builder.TaskEntryPoint = (typeof(
    TileSchedulerTask)).FullName;
builder.SetTrigger(new TimeTrigger(180, false));
builder.Register();
}
```

8. Add the following code snippet to the `OnNavigatedTo` method:

```
_timer.Start();
UpdateClockText();
CreateClockTask();
```

9. Add the following code snippet below the `MainPage` class definition:

```
public sealed class TileSchedulerTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        var deferral = taskInstance.GetDeferral();
        CreateSchedule();
        deferral.Complete();
    }

    public static void CreateSchedule()
    {
        var tileUpdater = TileUpdateManager.
            CreateTileUpdaterForApplication();
        var plannedUpdated = tileUpdater.
            GetScheduledTileNotifications();

        DateTime now = DateTime.Now;
        DateTime planTill = now.AddHours(4);

        DateTime updateTime = new DateTime(now.Year, now.Month,
            now.Day, now.Hour, now.Minute, 0).AddMinutes(1);
        if (plannedUpdated.Count > 0)
            updateTime = plannedUpdated.Select(x =>
                x.DeliveryTime.DateTime).Union(new[] { updateTime
            }).Max();
        XmlDocument documentNow = GetTilenotificationXml(now);

        tileUpdater.Update(new TileNotification(documentNow) {
            ExpirationTime = now.AddMinutes(1) });
    }
}
```

```

for (var startPlanning = updateTime;
    startPlanning < planTill; startPlanning =
        startPlanning.AddMinutes(1))
{
    Debug.WriteLine(startPlanning);
    Debug.WriteLine(planTill);

    try
    {
        XmlDocument document = GetTilenotificationXml(
            startPlanning);

        var scheduledNotification = new
            ScheduledTileNotification(document,
                new DateTimeOffset(startPlanning))
        {
            ExpirationTime = startPlanning.AddMinutes(1)
        };

        tileUpdater.AddToSchedule(scheduledNotification);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Error: " + ex.Message);
    }
}

private static XmlDocument GetTilenotificationXml(
    DateTime dateTime)
{
    string language =
        GlobalizationPreferences.Languages.First();
    var cultureInfo = new CultureInfo(language);

    string shortDate = dateTime.ToString(
        cultureInfo.DateTimeFormat.ShortTimePattern);
    string longDate = dateTime.ToString(
        cultureInfo.DateTimeFormat.LongDatePattern);

    var document = XElement.Parse(string.Format(@"<tile>
<visual>
    <binding template=""TileSquareText02"">
        <text id=""1"">{0}</text>
        <text id=""2"">{1}</text>
    </binding>
</visual>
</tile>
", shortDate, longDate));
}

```

```

        </binding>
        <binding template="TileWideText01">
            <text id="1">{0}</text>
            <text id="2">{1}</text>
            <text id="3"></text>
            <text id="4"></text>
        </binding>
    </visual>
</tile>", shortDate, longDate));

    return document.ToXmlDocument();
}
}

public static class DocumentExtensions
{
    public static XmlDocument ToXmlDocument(this
        XElement xDocument)
    {
        var xmlDocument = new XmlDocument();
        xmlDocument.LoadXml(xDocument.ToString());
        return xmlDocument;
    }
}

```

10. Run the program.

How it works...

The preceding program shows how to create a background time-based task and how to show the updates from this task on a live tile on the Windows 10 start menu. Programming Universal Windows Platform applications is quite a challenging task itself—you have to care about an application suspending/restoring its state and many other things. Here, we are going to concentrate on our main task, leaving behind the secondary issues.

Our main goal is to run some code when the application itself is not in the foreground. First, we create an implementation of the `IBackgroundTask` interface. This is our code, and the `Run` method will be called when we get a trigger signal. It is important that if the `Run` method contains asynchronous code with `await` in it, we have to use a special deferral object as shown in the recipe to explicitly specify when we begin and end the `Run` method execution. In our case, the method call is synchronous, but to illustrate this requirement, we work with the deferral object.

Inside our task in the `Run` method, we create a set of tile updates each minute for 4 hours and register it in `TileUpdateManager` with the help of the `ScheduledTaskNotification` class. A tile uses a special XML format to specify exactly how the text should be positioned in it. When we trigger our task from the system, it schedules one-minute tile updates for the next 4 hours. Then, we need to register our background task. We do this twice; one registration provides a `UserPresent` trigger, which means that this task will be triggered when a user is logged in. The next trigger is a time trigger, which runs the task once every 3 hours.

When the program runs, it creates a timer, which runs when the application is in the foreground. At the same time, it tries to register background tasks; to register these tasks, the program needs user permission, and it will show a dialog requesting permissions from the user. Now, we have scheduled live tile updates for the next 4 hours. If we close our application, the live tile will continue to show the new time every minute. In the next 3 hours, the time trigger will run our background task once again, and we will schedule another live tile update.

Running a .NET Core application on OS X

This recipe shows how to install a .NET Core application on OS X and how to build and run a .NET console application.

Getting ready

To go through this recipe, you will need a Mac OS X operating system. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter11\Recipe4`.

How to do it...

To understand how to run .NET Core applications, perform the following steps:

1. Install .NET Core on your OS X machine. You can visit <http://dotnet.github.io/getting-started/> and follow the installation instructions there. Since .NET Core is in the pre-release stage, the installation and usage scenarios could change before this book is published. Refer to the site instructions in that case.
2. After you have downloaded the `.pkg` file, hold the *Control* key while opening it. It will unblock the file and will allow you to install it.
3. After you have installed the package, you will need to install OpenSSL. The easiest way is to install the homebrew package manager first. Open the terminal window and run the following command:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

4. Then, you can install OpenSSL by typing the following in it:
`brew install openssl`
5. There is also the small catch that .NET Core at the time of writing needs to increase the open files limit. This can be achieved by typing the following:
`sudo sysctl -w kern.maxfiles=20480`
`sudo sysctl -w kern.maxfilesperproc=18000`
`sudo ulimit -S -n 2048`
6. Now you have installed .NET Core and are ready to go. To create a sample Hello World application, you can create a directory and create an empty application:
`mkdir HelloWorld`
`cd HelloWorld`
`dotnet new`
7. Let's check whether the default application works. To run the code, we have to restore dependencies and build and run the application. To achieve this, type the following commands:
`dotnet restore`
`dotnet run`
8. Now, let's try to run some asynchronous code. In the `Program.cs` file, change the code to the following:

```
using System;
using System.Threading.Tasks;
using static System.Console;
namespace OSXConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine(".NET Core app on OS X");
            RunCodeAsync().GetAwaiter().GetResult();
        }
        static async Task RunCodeAsync()
        {
            try
            {
                string result = await GetInfoAsync("Async 1");
                WriteLine(result);
                result = await GetInfoAsync("Async 2");
                WriteLine(result);
            }
        }
    }
}
```

```

    }
    catch (Exception ex)
    {
        WriteLine(ex);
    }
}
static async Task<string> GetInfoAsync(string name)
{
    WriteLine($"Task {name} started!");
    await Task.Delay(TimeSpan.FromSeconds(2));
    if (name == "Async 2")
        throw new Exception("Boom!");
    return
        $"Task {name} completed successfully!"
    // + $"Thread id {System.Threading.Thread.CurrentThread.
    ManagedThreadId}."
    ;
}
}
}

```

9. Run the program with the `dotnet run` command.

How it works...

Here, we download a `.pkg` file with the .NET Core installation package from the site and install it. We also install the OpenSSL library using the homebrew package manager (which also gets installed). Besides that, we increase the open files limit in OS X to be able to restore .NET Core dependencies.

Then, we create a separate folder for the .NET Core application, create a blank console application, and check whether everything works fine with restoring dependencies and running the code.

Finally, we create a simple asynchronous code and try to run it. It should run well, showing the messages that the first task completed successfully. The second task caused an exception, which was correctly handled. But if you try to uncomment a line that is intended to show the thread-specific information, the code will not be compiled, since .NET Core has no support for Thread APIs.

Running a .NET Core application on Ubuntu Linux

This recipe shows how to install a .NET Core application on Ubuntu and how to build and run a .NET console application.

Getting ready

To go through this recipe, you will need an Ubuntu Linux 14.04 operating system. There are no other prerequisites. The source code for this recipe can be found at `BookSamples\Chapter11\Recipe5`.

How to do it...

To understand how to run .NET Core applications, perform the following steps:

1. Install .NET Core on your Ubuntu machine. You can visit <http://dotnet.github.io/getting-started/> and follow the installation instructions there. Since .NET Core is in the pre-release stage, the installation and usage scenarios could change by the time this book is published. Refer to the site instructions in that case.

2. First, open a terminal window and run the following commands:

```
sudo sh -c 'echo "deb [arch=amd64] http://apt-mo.trafficmanager.net/repos/dotnet/ trusty main" > /etc/apt/sources.list.d/dotnetdev.list'

sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893

sudo apt-get update
```

3. Then, you can install .NET Core by typing the following in the terminal window:

```
sudo apt-get install dotnet=1.0.0.001331-1
```

4. Now, you have installed .NET Core and are ready to go. To create a sample Hello World application, you can create a directory and create an empty application:

```
mkdir HelloWorld
cd HelloWorld
dotnet new
```

5. Let's check whether the default application works. To run the code, we have to restore dependencies and build and run the application. To achieve this, type the following commands:

```
dotnet restore
dotnet run
```

6. Now, let's try to run some asynchronous code. In the `Program.cs` file, change the code to the following:

```
using System;
using System.Threading.Tasks;
using static System.Console;
namespace OSXConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine(".NET Core app on Ubuntu");
            RunCodeAsync().GetAwaiter().GetResult();
        }
        static async Task RunCodeAsync()
        {
            try
            {
                string result = await GetInfoAsync("Async 1");
                WriteLine(result);
                result = await GetInfoAsync("Async 2");
                WriteLine(result);
            }
            catch (Exception ex)
            {
                WriteLine(ex);
            }
        }
        static async Task<string> GetInfoAsync(string name)
        {
            WriteLine($"Task {name} started!");
            await Task.Delay(TimeSpan.FromSeconds(2));
            if (name == "Async 2")
                throw new Exception("Boom!");
            return
                $"Task {name} completed successfully!"
                // + $"Thread id {System.Threading.Thread.CurrentThread.
                ManagedThreadId}."
                ;
        }
    }
}
```

7. Run the program with `dotnet run` command.

How it works...

Here, we start with setting up the apt-get feed that hosts the .NET Core packages that we need. This is necessary since at the time of writing, .NET Core for Linux may not have been released. For sure, when the release happens, it will get into normal apt-get feeds and you won't have to add custom feeds to it. After completing this, we use apt-get to install the currently working version of .NET Core.

Then, we create a separate folder for the .NET Core application, create a blank console application, and check whether everything works fine with restoring dependencies and running the code.

Finally, we create a simple asynchronous code and try to run it. It should run well, showing messages that the first task completed successfully, and the second task caused an exception, which was correctly handled. But if you try to uncomment a line that is intended to show the thread-specific information, the code will not be compiled, since .NET Core has no support for Thread APIs.

Index

Symbols

.NET Core application

running, on OS X 237-239

running, on Ubuntu Linux 240-242

.NET thread pool 48

A

abstraction layer 67

APM pattern

converting, to task 75-78

AsOrdered method 151

asynchronous functions

about 94, 95

creating 94

asynchronous HTTP server and client

writing 187-189

asynchronous I/O operations

applications 181-183

asynchronous Observable

collection, converting to 162-165

asynchronous operations

creating, Rx used 177-180

exceptions, handling 104-107

posting, on thread pool 52, 53

asynchronous processing

generalizing, BlockingCollection

used 136-139

implementing, ConcurrentQueue

used 127-129

order, changing, ConcurrentStack

used 130-132

Asynchronous Programming Model (APM) 49

asynchronous task results

obtaining, with await operator 96-98

async operators

disadvantages 95

async void method

working with 111-114

atomic operations

about 28

performing 28-31

AutoResetEvent construct

using 34-36

await operator

disadvantages 95

dynamic type, using with 118-122

used, for obtaining asynchronous task results 96-98

used, for parallel asynchronous tasks execution 102-104

using, in lambda expression 98, 99

using, with consequent asynchronous tasks 100-102

B

BackgroundTask

using, in Universal Windows Platform applications 230-236

BackgroundWorker component

using 63-66

Barrier construct

using 39-41

basic operations

performing, with task 70-72

BlockingCollection

- about 124
- used, for generalizing asynchronous processing 136-139
- used, for implementing Parallel Pipeline 205-210

C

C#

- thread, creating 2-6

callback

- registering 58

cancellation option

- implementing 56-58

captured synchronization context

- use, avoiding 107-111

C# lock keyword

- used, for locking 19-21

closure mechanics 53

coarse-grained locking 127

collection

- converting, to asynchronous Observable 162-165

Common Language Runtime (CLR) 48

Compare and Swap (CAS) 124

ConcurrentBag

- used, for creating scalable crawler 132-136

ConcurrentDictionary

- about 124
- using 125-127

ConcurrentQueue

- used, for implementing asynchronous processing 127-129

ConcurrentStack

- used, for changing asynchronous processing order 130-132

consequent asynchronous tasks

- await operator, using with 100-102

context switch 28

continuation 75

CountDownEvent construct

- using 38, 39

custom aggregator

- creating, for PLINQ query 157-159

custom awaitable type

- designing 114-117

custom Observable

- writing 165-168

D

database

- working with, asynchronously 190-193

data parallelism 142

data partitioning

- managing in PLINQ query 153-157

degree of parallelism

- and thread pool 54-56

delegate

- about 48
- invoking, on thread pool 49-51

double-checked locking pattern 204

dynamic type

- using, with await operator 118-121

E

EAP pattern

- converting, to task 79, 80

Enqueue method 124

Event-based Asynchronous Pattern (EAP) 65

event handlers 65

events 65

exceptions

- handling 24-26
- handling, in asynchronous operations 104-107
- handling, in PLINQ query 151-153

F

files

- working with, asynchronously 183-186

fine-grained locking technique 127

First In, First Out (FIFO) 124

G

Gutenberg

- website link 219

H

hybrid constructs 28

I

I/O threads 48

iterators 95

K

kernel-mode constructs 28

L

lambda expression

about 50

await operator, using 98, 99

Last In, First Out (LIFO) 124

Lazy-evaluated shared states

implementing 200-204

LazyInitializer.EnsureInitialized method 204

LINQ queries

using, against observable

collection 174-176

LINQ query

parallelizing 145-148

M

ManualResetEventSlim construct

using 36-38

Map/Reduce pattern

about 200

implementing, with PLINQ 215-219

monitor construct

locking with 22-24

Mutex construct

using 31, 32

O

observable collection

LINQ queries, using against 174-176

Observable object

creating 172-174

OS X

.NET Core application, running 237-239

P

parallel asynchronous tasks

executing, await operator used 102-104

Parallel class

using 143-145

Parallel Framework Extensions (PFX) 141

Parallel Pipeline

about 200

implementing, with

BlockingCollection 205-210

implementing, with TPL DataFlow 210-214

parameters

passing, to thread 16-18

PLINQ

used, for implementing

Map/Reduce 215-219

PLINQ query

custom aggregator, creating 157-160

data partitioning, managing 153-157

exceptions, handling 151-153

parameters, tweaking 148-151

pooling 47

pull-based approach 161

push-based approach 162

R

Reactive Extensions (Rx)

about 161, 162

used, for creating asynchronous

operations 177-180

ReaderWriterLockSlim construct

using 41-43

S

scalable crawler

creating, ConcurrentBag used 132-136

SemaphoreSlim construct

using 32-34

shared-state object 199

Simple Object Access Protocol (SOAP) 197

SpinWait construct

using 44, 45

structured parallelism 142

Subjects type

using 168-171

T

task

- about 68
- APM pattern, converting to 75-78
- basic operations, performing with 70-72
- combining 72-75
- creating 69, 70
- EAP pattern, converting to 79, 80
- exception handling 83, 85
- running, in parallel 85-87

task execution

- tweaking, with TaskScheduler 87-91

task parallelism 141

Task Parallel Library (TPL) 51

task scheduler 70

thread

- aborting 8-10
- background 14, 15
- cancellation option, implementing 56-58
- creating, in C# 2-5
- foreground 14, 15
- making, wait 7, 8
- parameters, passing 16-18
- pausing 6, 7
- priority 12-14
- state, determining 10, 11

thread pool

- and degree of parallelism 54-56
- asynchronous operation, posting 52, 53
- delegate, invoking 49-51
- timeout, using 59-61
- wait handle, using 59-61

thread synchronization

- about 27
- AutoResetEvent construct 34-36
- Barrier construct 39-41
- CountDownEvent construct 38, 39
- ManualResetEventSlim construct 36-38
- Mutex construct 31, 32
- ReaderWriterLockSlim construct 41-43
- SemaphoreSlim construct 32-34
- SpinWait construct 44, 45

timeout

- using, with thread pool 59-61

timer

- using 61-63
- using, in Universal Windows Platform application 223-227

TPL DataFlow

- about 200
- used, for implementing, Parallel Pipeline 210-214

TryDequeue method 124

TryPeek method 124

U

Ubuntu Linux

- .NET Core application, running 240-242

Universal Windows Platform application

- BackgroundTask, using 230-236
- timer, using 223-227

unstructured parallelism 142

user-mode constructs 28

W

wait handle

- using, with thread pool 59-61

Windows Communication Foundation (WCF) service

- about 183
- calling, asynchronously 194-198

WinRT

- about 221
- using, from usual applications 227-230

WithExecutionMode method 151

WithMergeOptions method 151

worker thread 48

