



Quick answers to common problems

Visual Studio 2012 and .NET 4.5 Expert Development Cookbook

Over 40 recipes for successfully mixing the powerful capabilities
of .NET 4.5 and Visual Studio 2012

Abhishek Sur

[PACKT] enterprise 
PUBLISHING professional expertise distilled

www.allitebooks.com

Visual Studio 2012 and .NET 4.5 Expert Development Cookbook

Over 40 recipes for successfully mixing the powerful capabilities of .NET 4.5 and Visual Studio 2012

Abhishek Sur



BIRMINGHAM - MUMBAI

Visual Studio 2012 and .NET 4.5 Expert Development Cookbook

Copyright © 2013 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: April 2013

Production Reference: 1050413

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.
ISBN 978-1-84968-670-9

www.packtpub.com

Cover Image by Siddhart Ravishankar (sidd.ravishankar@gmail.com)

Credits

Author

Abhishek Sur

Project Coordinator

Anish Ramchandani

Reviewers

Carlos Hulot

Ahmed Ilyas

Sergiy Suchok

Ken Tucker

Proofreader

Claire Cresswell-Lane

Indexer

Tejal Soni

Acquisition Editor

Kartikey Pandey

Graphics

Aditi Gajjar

Lead Technical Editor

Susmita Panda

Production Coordinator

Aparna Bhagat

Technical Editors

Jalasha D'costa

Amit Ramadas

Cover Work

Aparna Bhagat

About the Authors

Abhishek Sur is a Microsoft MVP in Client App Dev since 2011. He is an architect in the .NET platform. He has profound theoretical insight and years of hands on experience in different .NET products and languages. Over the years, he has helped developers throughout the world with his experience and knowledge. He owns a Microsoft User Group in Kolkata named **KolkataGeeks**, and regularly organizes events and seminars in various places for spreading .NET awareness. A renowned public speaker, voracious reader, and technology buff, his main interest lies in exploring the new realms of .NET technology and coming up with priceless write-ups on the unexplored domains of .NET. He is associated with the Microsoft Insider list on WPF and C#, and is in constant touch with product group teams. He holds a Masters degree in Computers along with various other certificates to his credit.

On the web, Abhishek is a freelance content producer, developer, and a site administrator. His website abhisheksur.com guides both budding and experienced developers to understand the details of languages and latest technology. He enjoys a huge fan following on social networks. You can reach him at books@abhisheksur.com, or get online updates from Facebook or Twitter @abhi2434.

Acknowledgement

Writing this book would not have been possible without the help of many people. Knowledge is a priceless alternative of time. While writing this book, I have spent a considerable amount of time in reading a lot of online journals, documentations, and blogs to explain and present concepts and ideas in a more lucid way.

First of all, I am extremely thankful to the entire Packt Publishing team for their continuous help. I would like to thank Anish Ramchandani and Susmita Panda, who have been very supportive and helped me to meet the deadlines. My most sincere thanks to my reviewers Carlos Hulot, Ahmed Ilyas, Sergiy Suchok, and Ken Tucker for reviewing every line I wrote for this book, which resulted in bringing so many positive changes in this book. Last but not the least, a lot of thanks to Amit Ramadas and Jalasha D'costa for editing the chapters of this book.

I would also like to thank my colleague cum friends Shibatosh, Ranjit, Pallab, Ayan, Malini, and other members of our group for their continuous support and motivation to write this book. I am also thankful for being in touch with Mr. Raj Goswami, CEO and Mr. D. K Goswami, Director of BuildFusion, India. I would also like to extend my sincere thanks to Anoop Madhusudan, Abhijit Jana, Kunal Chowdhury, Dhananjay Kumar, Karthikayan Anabarasana, Lohith, Abhishek Kant, and Shivprasad Koirala for their continuous help and motivation.

I would also like to acknowledge my fiancée Riya for helping me and motivating me to write better. I want to dedicate this book to her.

Being a community lover I used to write a lot of blogs and articles, but this is the first time I am putting my experience and efforts in the form of a book. I am really thankful to all my readers and followers all over the world who loved my blogs and hopefully they will love this book too.

About the Reviewers

Carlos Hulot has been working in the IT area for more than 20 years in different capabilities, from software development, project management to IT marketing product development and management. Carlos has worked for multinational companies like Royal Philips Electronics, PricewaterhouseCoopers, and Microsoft. Currently, Carlos is working as an independent IT consultant. Carlos is a Computer Science lecturer at two Brazilian universities. Carlos holds a Ph.D in Computer Science and Electronics from the University of Southampton, UK, and a B.Sc. in Physics from the University of São Paulo, Brazil.

Ahmed Ilyas has a bachelor's degree in engineering from Napier University in Edinburgh, Scotland, and has majored in software development. He has 15 years of professional experience in software development.

After leaving Microsoft, he ventured into setting up his consultancy company offering the best possible solutions for a magnitude of industries and providing real-world answers to problems. They only use the Microsoft stack to build these technologies, to be able to bring in best practice, patterns, and software to their client base. Thus, enabling long term stability and compliance in the ever changing software industry and also improving software developers around the globe—pushing the limits in technology as well as develop themselves to become better.

This went on to being awarded the MVP in C# three times by Microsoft for providing excellence and independent real-world solutions to problems that developers face.

With the breadth and depth of knowledge he as obtained not only from his research but also with the valuable wealth of information and research at Microsoft, the motivation and inspirations come from this, with 90 percent of the world using at least one form of Microsoft technology.

Ahmed Ilyas has worked for a number of clients and employers. With the great reputation that he has, this has resulted in having a large client base for his consultancy company, Sandler Ltd (UK) which includes clients from different industries from, media to medical and beyond. Some clients have included him on their "approved contractors/consultants" list, which include ICS Solution Ltd and he has been placed on their "DreamTeam" portal and also CODE Consulting/EPS Software (www.codemag.com, based in the US).

Ahmed Ilyas has also been involved, in the past, in reviewing books for Packt Publishing and wishes to thank them for the opportunity once again.

I would like to thank the author/publisher of this book for giving me the great honor and privilege in reviewing the book. I would also like to thank my client base and especially Microsoft Corporation and my colleagues over there for enabling me to become a reputable leader as a software developer in the industry, which is my passion.

Sergiy Suchok graduated in 2004 with honors from the Faculty of Cybernetics, Taras Shevchenko National University of Kyiv (Ukraine), and has since then been keen on information technology. He currently works in the banking area and has a PhD in Economics. Sergiy is the coauthor of more than 45 articles and has participated in more than 20 scientific and practical conferences devoted to economic and mathematical modeling. He is a member of the New Atlantis Youth Public Organization (newatlantida.org.ua) and devotes his leisure time to environmental protection issues, historical, and patriotic development and popularization of a grateful attitude towards the Earth. He also writes poetry and short stories and makes macramé.

I would like to thank my wife and my young daughter for their patience and understanding while reviewing.

Ken Tucker is a web developer for Sea World, and has been a Microsoft MVP since October 2003. In his spare time he enjoys writing Windows Phone and Windows Store apps.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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 service@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.



<http://PacktLib.PacktPub.com>

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

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: Introduction to Visual Studio IDE Features	7
Introduction	7
Identifying the various components of Visual Studio IDE	8
Working with Solution Explorer and Class View	14
Working with the main workspace area of IDE	18
Navigating between code inside the IDE	22
Extending Visual Studio templates	29
Using Code Snippets in Visual Studio	39
Using Smart Tags and Refactor in Visual Studio	45
Chapter 2: Basics of .NET Programs and Memory Management	51
Introduction	52
Inspecting the internal structure of a .NET assembly	53
Working with different types of assemblies	60
Inspecting the major components of a .NET program	70
How to work with custom configurations for an application	75
How to disassemble an assembly	81
Securing your code from reverse engineering by using obfuscation	89
Understanding .NET garbage collection and memory management	96
How to find memory leaks in a .NET program	104
Solutions to 10 common mistakes made by developers while writing code	115
Chapter 3: Asynchronous Programming in .NET	123
Introduction	123
Introduction to Threading and Asynchronous Threading patterns	125
Working with Event-based asynchronous pattern and BackgroundWorker	135
Working with thread locking and synchronization	139
Lock statement using task-based parallelism in concurrent programming	147

Working with async and await patterns	155
Working with Task Parallel Library data flows	165
Chapter 4: Enhancements to ASP.NET	173
Introduction	173
Understanding major performance boosters in ASP.NET web applications	174
How to work with statically-typed model binding in ASP.NET applications	185
Introduction to HTML5 and CSS3 in ASP.NET applications	189
Working with jQuery in Visual Studio with ASP.NET	205
Working with task-based asynchronous HttpHandlers and HttpModules	211
New enhancements to various Visual Studio editors	214
Chapter 5: Enhancements to WPF	223
Introduction	223
Getting started with WPF and its major enhancements in .NET 4.5	226
Building applications using MVVM pattern supported by WPF	234
Using the Ribbon User Interface in WPF	260
Using WeakEvent pattern in WPF	272
Chapter 6: Building Touch-sensitive Device Applications in Windows 8	275
Introduction	275
Building your first Windows 8 style tiles application using JavaScript, HTML5, and CSS	278
Writing a library for WinJS	294
Building your first Windows 8 style tiles application using C# and XAML	297
Working with storage files in Windows 8 style tiles applications	306
Understanding the application life cycle of WinRT applications	312
Chapter 7: Communication and Sharing using Windows 8	325
Introduction	325
How to enable app to app sharing inside a Windows 8 environment	326
Working with notification and services	330
How to perform background transfers of data in Windows 8 style tiles applications	348
Index	355

Preface

Moving through the last few decades, from being the new kid on the block, .NET has turned itself into the most talked about young thing on the IT horizon. More and more people are getting inclined to use .NET, to make a mark in their career and also, more and more, clients are coming to .NET to accomplish their dreams. The recent buzz about C# being the best programming language has now made the world of developers very eager to build .NET applications. As the crowd is growing in the developers arena, there is always a need to have a clear perception on how things work and ways to make things better.

Visual Studio 2012 and .NET 4.5 Expert Development Cookbook mainly focuses on things that you need to know in those crunch situations as a developer. It also tries to feed you with as much details as it can, and covers as much technological domain in the world of .NET. The book is written in the form of recipes with step-by-step tutorials on every topic where the developer accompanies the author in this wonderful journey into the known and hitherto unknown realms of .NET. The recipes in the book, which are mostly sought out by developers on the Internet, are chosen in such a way so as to practically demonstrate them and not restrict developers only to them, but to also expose the other unexplored domains on the same topic to give them a clear view of the whole picture. There is a special section for each recipes bearing the heading *There's more...*, which always focuses on giving you extra knowledge on things that you might have missed without it. By the time you come to the end of this journey, you will feel the comfort and enjoy the confidence that a clear understanding of the insight of .NET gives you.

The book is a practical handbook that could give you optimal utilization of time for knowledge. It separately presents the topics very precisely and elaborates the same which you can rely on for a deeper look. It explains the recipes with proper sample code blocks that might make the usage of each topic very clear and make you utilize the final source code while writing your real-world applications. The examples taken for this book will clear your understanding of how things exactly work for that particular recipe and also adapt you as a developer to make use of the same source code in your production environment efficiently and quickly. If you want to utilize your busy schedule to explore all the necessary ongoing technology in the market, this book is best suited for you.

The book focuses on giving you:

- ▶ Maximum utilization of time for learning
- ▶ Major insights into ongoing technologies such as Visual Studio 2012, .NET 4.5, ASP.NET, Windows 8 Applications, Windows Presentation Foundation, HTML5, jQuery, memory management, and so on
- ▶ Practical examples on procedures to create real-world applications
- ▶ Step-by-step examples to create simple applications based on heads

What this book covers

Chapter 1, Introduction to Visual Studio IDE Features, starts with a basic introduction to Visual Studio IDE and gives the developer insights into how to increase productivity of development using a common set of tools and features present inside the IDE.

Chapter 2, Basics of .NET Programs and Memory Management, introduces the intersection of a .NET program and its core components. It dives deep in demonstrating the .NET infrastructure with detailed explanation of memory management and related techniques.

Chapter 3, Asynchronous Programming in .NET, focuses on introducing all existing techniques to deal with threading in .NET followed by the newer patterns that takes over the existing working principles with in-depth explanation on their working principles.

Chapter 4, Enhancements to ASP.NET, gives you an introduction to latest enhancements of ASP.NET 4.5 with HTML5 and jQuery. It also introduces some of the performance boosters available in .NET 4.5 and Visual Studio 2012 with ASP.NET.

Chapter 5, Enhancements to WPF, introduces the enhancements to WPF 4.5 and the major components of WPF. It gives a practical implementation of MVVM based WPF application covering all the facets required to program in WPF environment.

Chapter 6, Building Touch-sensitive Device Applications in Windows 8, introduces the new programming model for developing Windows 8 style tiles application. It gives a step-by-step introduction in how to program using HTML5 and JavaScript as well as WPF and C# for developing Windows 8 applications.

Chapter 7, Communication and Sharing Using Windows 8, focuses on how to implement network-enabled applications in Windows 8 with step by step implementation on how sharing and searching works inside the Windows 8 environment.

Appendix, .NET languages and its Construct, focuses on giving insights on how languages work in the .NET framework and C# with details explanation with examples of various features of C# language.

You can download this Appendix from http://www.packtpub.com/sites/default/files/downloads/6709EN_Appendix_NET_Languages_and_its_Construct.pdf

What you need for this book

The basic software requirements for this book are as follows:

- ▶ Microsoft .NET Framework 4.5 and higher
- ▶ Microsoft Visual Studio 2012 Express or higher editions
- ▶ Windows 8 Operating System (especially to work with Chapter 6 and 7)
- ▶ Latest web browsers

Who this book is for

The purpose of this book is to give you ready made steps in the form of recipes to develop common tasks that, as a developer, you might often be required to access. The book utilizes its chapters skillfully to provide as much information as it can and also with as much detail as necessary to kick start the subject. The book also delivers in-depth analysis of some advanced section of the subject to get you to expertise level. If you are a starter in the development environment and want to get expertise on ongoing technologies in the market, this book is ideal for you. Even for architects and project managers, the book can be a guide to enrich their existing knowledge.

The book uses C# and Visual Studio 2012 with Windows 8 (as the operating system) in the examples. Even though the book does not require any knowledge to start, it expects some basic theoretical and practical overall experience on the subjects to understand the recipes. The book bridges the gap between a normal developer to an expert architect.

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: "The `requestValidationMode` attribute of `httpRuntime` defines how the request is validated to the server before calling `HttpPipeline`."

A block of code is set as follows:

```
(function($) {  
    $.fn.extend({  
        Value1 : 20,  
        myMethod : function(msg) {  
            alert(msg + "value : " + this.Value1);  
        }  
    });  
})(jQuery);
```

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

```
Install-Package Microsoft.Web.Optimization
```

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: "you can also open Nuget package manager by right-clicking on the references folder of the project and select **Add Library Package Reference**".



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 all Packt books you have purchased 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.

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. You can also contact the author directly at books@abhisheksur.com to address any technical problems while covering the recipes.

1

Introduction to Visual Studio IDE Features

In this chapter, we will start with a basic introduction to Visual Studio IDE, and understand how we can increase the productivity of our development using some of the tools and features present in the IDE. After going through the chapter, you will understand the following recipes:

- ▶ Identifying the various components of Visual Studio IDE
- ▶ Working with Solution Explorer and Class View
- ▶ Working with the main workspace area of IDE
- ▶ Navigating between code inside the IDE
- ▶ Extending Visual Studio templates
- ▶ Using Code Snippets in Visual Studio
- ▶ Using Smart Tags and Refactor in Visual Studio

Introduction

Ever since Microsoft announced .NET for the first time almost 10 years ago, there has been a lot of noise in the developer community about the way the changes are going. .NET led its way to modernize the ideas of coding with more sophisticated techniques by adopting more object-oriented paradigm in programming and also changing the style of coding altogether. The Microsoft forerunner VB was announced to be modernized in the new environment and redesigned to be named as VB.NET, and also some other languages that are totally different in syntax, such as C#, J#, and C++ have been announced. All of these languages are built on top of the .NET Runtime (known as **Common Language Runtime** or **CLR**) and produce the same intermediate output in **Microsoft Intermediate Language (MSIL)**.

Microsoft announced .NET runtime as a separate entity by defining standardized rules and specifications that every language must follow to take advantage of CLR. The entirely new set of libraries, classes, syntaxes, or even the way of coding in Microsoft technologies, created a huge hindrance in the developer community. Many developers switched their jobs, while there are a few who really switched gears to understand how to work with the new technology that is totally different from its predecessors. The community has already started to realize that the existing set of Microsoft tools might not satisfy the needs of new evolving technology. Microsoft had to give a strong toolset to help the developers to work easier and better with the new technology.

Visual Studio is the answer to some of them. Microsoft Visual Studio is an **Integrated Development Environment (IDE)** to work with Microsoft languages. It is the premier tool that developers can possess to easily work with Microsoft technologies. But you should note, Visual Studio is not a new product from Microsoft. It has been around for quite sometime, but the new Visual Studio had been redesigned totally and released as Visual Studio 7.0 to support .NET languages.

Evolution of Visual Studio

As time progressed, Microsoft released newer versions of Visual Studio with additional benefits and enhancements. Visual Studio being a plugin host to host number of services as plugins, has evolved considerably with a lot of tools and extensions available; it has been the integral part of every developer's day-to-day activity. Visual Studio is not only a tool used by developers, but it has been identified that a large number of people who are not a part of the developer community have been loving this IDE and using it for editing/managing documents. The wide acceptance of Visual Studio to the community had made the product even better.

This year, Microsoft has released the latest version of Visual Studio. In this chapter, we will tour Visual Studio IDE features, its utilities, and mostly cover parts that can really help to make your work done more quickly.

Identifying the various components of Visual Studio IDE

Visual Studio 2012 has come up with lots of new enhancements and features. Some of these features widely enhance productivity of development. Knowing your IDE better is always an advantage to a developer. In this recipe, we will try to get our hands on to various Visual Studio IDE features to get started with using Visual Studio.

Getting ready

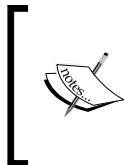
Before we start using Visual Studio, we need to first make a choice on which version practically suits us. Let's have a look at the features of all the versions of Visual Studio.

- ▶ **Visual Studio Express:** If you are looking to try out small applications or medium-sized applications and do not want to spend a single penny from your pocket, Visual Studio Express is the right choice for you. Microsoft has given the Express build free to everyone that is capable of doing all the basic needs of software build up.
- ▶ **Visual Studio Professional:** This edition of Visual studio is for individual development with most of the important debugging tools and all the things a developer commonly needs. So if your primary orientation of using the IDE is basic development, this would be the right choice for you. This edition is reasonable in price too.
- ▶ **Visual Studio Premium:** Visual studio Premium edition is for people who make high-quality usage of the IDE. It adds tools for testing, code analysis, debugging, profiling, discovers common coding errors, generate test data, and so on.
- ▶ **Visual Studio Ultimate:** This is the ultimate edition of the product with all the components that could exist within Visual Studio. This edition provides advanced debugging capabilities with all architecture and modeling tools with it.

You can find the entire comparison list between all the versions of Visual Studio from the link below:

<http://www.microsoft.com/visualstudio/eng/products/compare>

Once you are determined on what suits your requirement best, you can install it on your machine and we are ready to go.



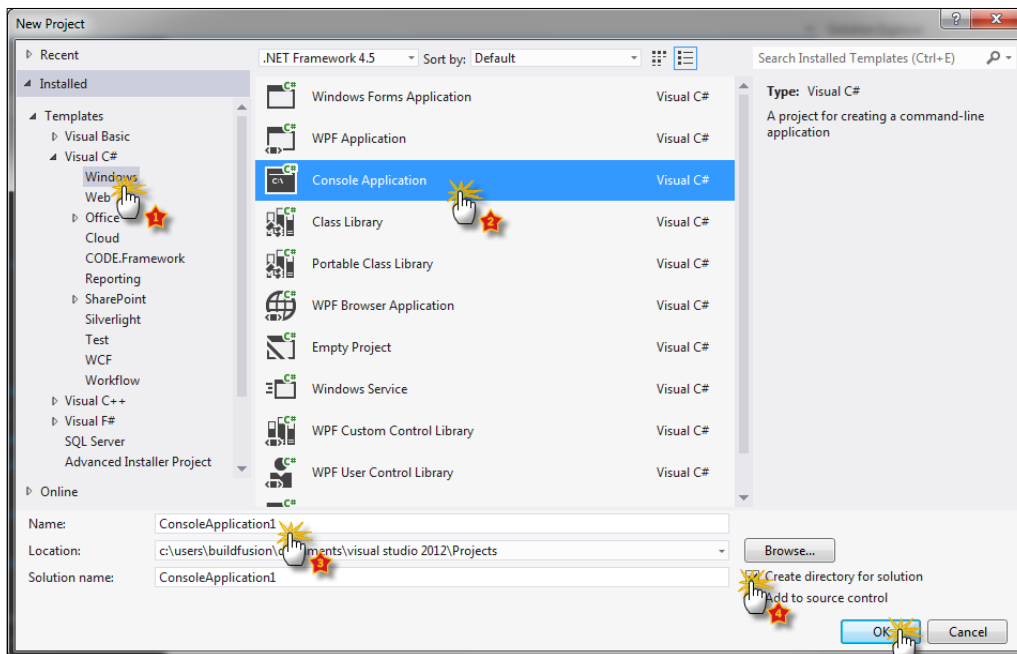
If you are opening the IDE for the first time, the IDE will present you few options and ask you what type of development you want to take. In most cases, I would recommend you to choose **General Development**, as this is the most convenient layout of Visual Studio.

How to do it...

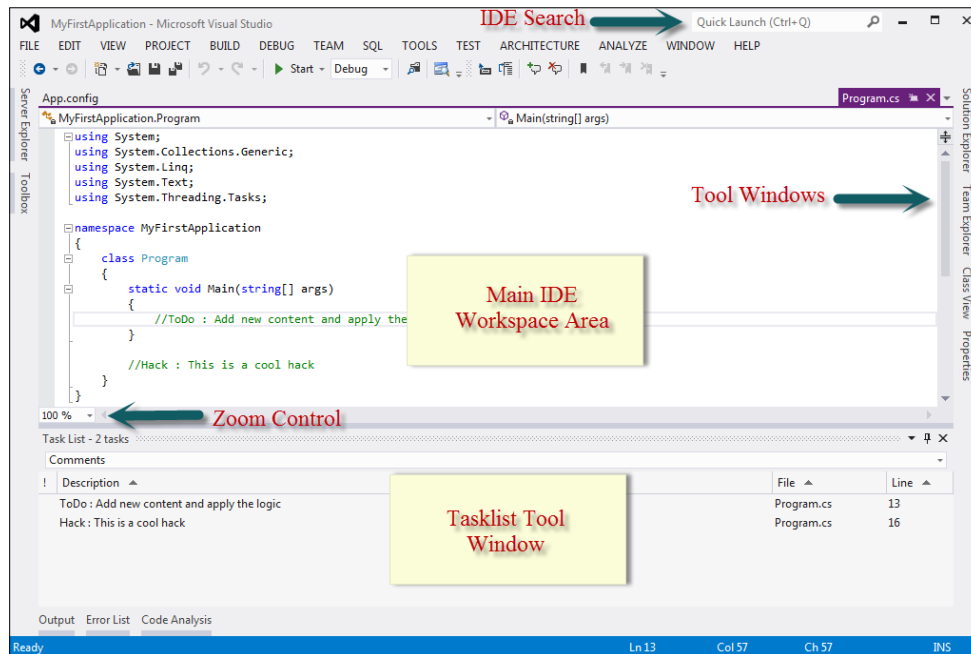
In this recipe, we will understand the different sections of the Visual Studio 2012 IDE and will show you where to start.

1. To start with the recipe, let us navigate to the **Start** menu | **All Programs**, choose the **Visual Studio 2012** folder and select **Visual Studio 2012**. This will launch the Visual Studio IDE.

2. After displaying the initial splash screen for a while when the IDE has been loaded, it presents you with a **Start** page with a three main options:
 - ❑ **Connect to Team foundation server**
 - ❑ **New Project**
 - ❑ **Open Project**
3. We can use either **New Project** here from the link, or we can navigate to **File | New Project** to create a new project. This pops up a **New Project** dialog box. In this dialog box, we have a number of options available based on the packages that are currently installed with the IDE. On the left-hand side of the dialog box, we see a tree of various items installed within the IDE. You can see there are a number of templates listed in the tree. When one item on the left-hand side gets selected, the corresponding templates associated with that group will be listed in the middle section of the dialog box (marked as **2** in the next screenshot).
4. Select an appropriate name for your project in the **Name:** field (marked as **3** in the next screenshot).
5. You can select the **Create directory for the solution** checkbox (marked as **4** in the next screenshot) to indicate that a new folder with the name just specified will be created inside the location you specify, which will hold the various files rather than storing them directly inside the specified location.



6. We choose **Visual C#** from the left-hand side pane, **Console Application** from the middle pane; keeping the default name we click on **OK** as shown in the previous screenshot. If everything is good, it opens the IDE and displays something as shown in the following screenshot:



7. In the previous screenshot, we have marked a few sections of the IDE which need special attention. They are as follows:
- ❑ The first section is **IDE Search**, which is just a blank textbox to search the IDE component.
 - ❑ **Tool Windows** are docked on the left, right, or bottom of the screen. When a tool window is open as shown in **TaskList Tool Window** at the bottom, it shows up a small dockable container and when it is collapsed, it shows a reference of it in the IDE sidebar as shown in the left-hand and right-hand side of the window.
 - ❑ The main IDE workspace area represents the main working area of the IDE. This forms the major portion of the IDE and mainly the application developer writes code here.
 - ❑ A special **Zoom Control** is also there inside the IDE, which helps to zoom in and out of the editor.
8. Finally, you can start writing your code in the main working area of the IDE or start exploring other options in the IDE yourself.

How it works...

There are a few things that need attention when a Visual Studio IDE is opened. Visual Studio is a process that is launched using an executable called `devenv` (which can be spelled as **Developer's Environment**). You can either double-click on the Visual Studio icon from the **Start** menu (which most of the people do), or go to **Start** and then search for `devenv` to run the IDE. The IDE is generally invoked in default permission mode. Sometimes, it is important to open the IDE as **Administrator** to enjoy administrative features on the environment. To change this behavior, you can right-click on the shortcut and select **Run as Administrator**. You can also permanently set the IDE to run as administrator from the **Properties** menu.

After the Visual Studio initial splash screen is displayed during the opening sequence, the first thing that you see is the **Start** page. We have navigated to **File | New Project** to open the **New Project** dialog box. As shown in the first screenshot, on the left-hand side of the window (marked as **1**), we see a tree of all the installed project type groups into collapsible panels.

If you do not find your template, you can also use **Search Installed Template** to search any template by its name in the right-hand corner of the dialog box.

As more than one framework can coexist in the same PC, the **New Project** dialog box is smart enough to allow you to choose the Framework that you need to use while deploying the application. By default it shows .NET 4.0 as the framework for the project, but you can change it by selecting the dropdown. The whole environment will change itself to give you only the options available for your current selection.

We choose **Visual C#** from the left tree and select **Console Application** from the middle pane as project template. Upon choosing any template, the description of the current template is loaded on the right-hand side of the screen. It gives you a brief idea on what **Console Application** is and is capable of doing.

At the bottom, we have the option to name the project and the solution, and we also have option to select the location where the project needs to be created (marked as **3**). You can select your own folder path to store the files you create inside the project by choosing the appropriate filesystem path in the box.

There are two checkboxes available as well. One of them is **Create directory for solution**, when selected (which is by default remains selected) creates a directory below the chosen path and places the files inside it. Otherwise it will create files just inside the folder chosen. To make it a habit, it is good to keep it selected.

Finally, click on **OK** to create the project with default files.

After the project is created, the basic IDE you see looks like the screenshot in step 5. We will now divide the whole IDE into those parts and explore the IDE together in the recipes that follow.

Let's paste the code inside the `Main` method that you see when you open the program class and paste the following code between the curly braces of `Main` method:

```
string content = "This is the test string to demonstrate Visual Studio
IDE features. ";
string content2 = "This is another string content";
Debug.Assert(content.Equals(content2), "The contents of the two
strings are not same");
Console.WriteLine("Thanks!");
Console.ReadKey(true);
```



Downloading the example code

You can download the example code files for all Packt books you have purchased 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.

After you paste the code, let's press *F5* on the keyboard. You will see a **Console** window that appears inside the IDE showing a message. Press the *Enter* key to close the **Console**.

There's more...

There are lots of other options that Visual Studio comes up with. In spite of opening Visual Studio normally, you can also use some special options to handle Visual Studio better. Let's try to look into other options that can be good to eye on.

Visual Studio command switches

Visual Studio being a normal executable that runs under Windows also provides some switches that can be used when we open the IDE. To use Visual Studio with these switches, we need to either use command prompt or use **Run** to add switches to the IDE. To use command prompt, just navigate to the location `%systemdrive%\Program Files\Microsoft Visual Studio 11\VC` and type in `devenv /?` to get a list of all the command switches available for the IDE. For instance `devenv /resetsettings`.

This command switch will reset all the user settings that have been applied to the IDE. The reset settings can also be used to specify the `vssettings` file, which has been used up to override settings for the current IDE. Similarly, you can use `devenv /resetSkipPkgs`.

This command will reset loading of all user-related tags associated to the packages that need to load to make everything work smoothly with Visual Studio. Sometimes if the IDE gets corrupted or loading time increases, you can also turn on diagnostic load using `devenv /SafeMode`.

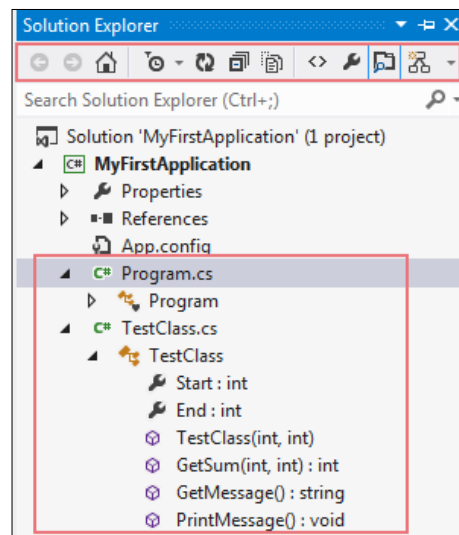
You can also try running a project without opening the IDE totally. I mean if you only need to open the IDE, run the project, and exit. The best option is to use Command switch `devenv /runexit "[solution/Project Path]"`.

You need to replace the `[solution/Project path]` with the path where you find the Solution file (`.sln` extension) or project files (`.csproj/.vbproj` files).

To see all the command switches supported by the executable, you can try `devenv /?` from the command prompt too.

Working with Solution Explorer and Class View

The most important part of the IDE that you need most often is your **Solution Explorer**. **Solution Explorer**, which resides on the right-hand side of the IDE, is the most widely used navigation tool between files and classes. It is shown in the following screenshot:



In the above screenshot, you see the basic structure of **Solution Explorer** when the IDE is loaded with a project. The **Solution Explorer** window starts with the `Solution` file and loads all the projects that are associated with the solution in a tree. The very next node of the `Solution` file generally is the project file. For the sake of identifying each of the files in the IDE, it provides you the proper icon and also makes the project file names bold.

How to do it...

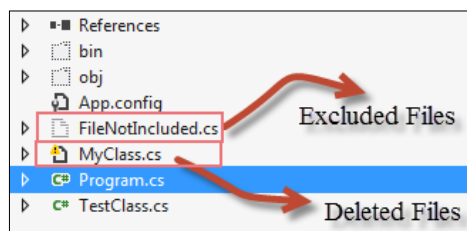
In this recipe, we are going to explore **Solution Explorer**.

1. On the right-hand side of the IDE, you will see **Solution Explorer** as shown in the preceding screenshot. This is the main screen area where you can interact with the files and folders associated with your project. If you do not see **Solution Explorer**, please navigate to **View | Solution Explorer** from the menu or press *Ctrl + W, S* from the keyboard.
2. Once you see the **Solution Explorer**, it should contain a tree of all the files and folders that are associated with the project. Toggle each node in the tree to see its related information. You can see in the figure, the **Solution Explorer** is capable of showing the list of members of a type that is written inside a file as well. The node **TestClass**, when opened, shows a tree of all its members in subsequent nodes.
3. The header section of **Solution Explorer** contains a number of buttons. These buttons are commands associated with the current selection of the tree node.

How it works...

Solution Explorer is the main window that lists the entire solution that is loaded to the IDE. It gives you an organized view of projects and files that are associated with the solution for easy navigation in a form of a tree. The outermost node of the **Solution Explorer** is the *Solution* itself, and below it are the projects, then files/folders. The *Solution* file also supports you to load folders directly inside the solution and even store documents in the first level. The project that is set as startup is marked in bold.

There are a number of buttons stacked at the top of the **Solution Explorer** window called toolbar buttons, and based on the type of file that is selected in the tree will be made available or disabled. Let's talk about each of them individually:



The solution tree in Visual Studio 2012 also loads the entire structure of the class into its nodes. Just expand the *.cs* file and you will see all its members and classes are listed. Visual Studio also has a class view window, but **Solution Explorer** is smart enough to list all the Class View elements inside its own hierarchy. You can open Class View by navigating to **View | ClassView** or pressing *Ctrl + W, C*, to see only the portion of class and its members.

Another important consideration is **Solution Explorer** as it shows the files from the Solution file, it also tracks the actual existence of the file in the physical locations too. While loading the files sometimes, it might show exclamatory signs if the file doesn't exists in physical location.

Here the MyClass file, even though it is a .cs file, does not show up the usual icon, but shows one exclamatory sign which indicates that the file is added to the solution, but the physical file does not exists.

On the contrary, some files are shown in the solution as blank files, (in our case FileNotIncluded.cs or folders like bin/obj). These files, even though they exist in the filesystem, are not included in the solution file.

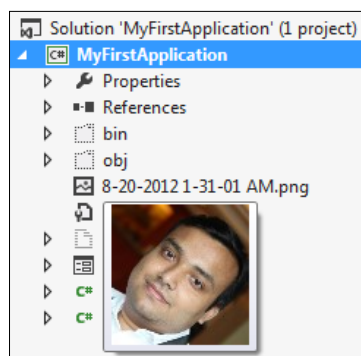
Each of the files show one **Additional Information** button on the right-hand side of the tree node in the solution. This button gives extra information associated with the file. For instance, if you click on the button corresponding to a .cs file it will pop up a menu with **Contains**. This will get the associated class view for the particular file in the solution. The menu can be pretty long depending on the items that cannot be shown in generalized toolbar buttons. When the solution loads additional information, there are forward/back buttons which can be used to navigate between views in the solution.

There's more...

In addition to the basic updates to **Solution Explorer**, there are lots of other enhancements that are made to the **Solution Explorer** to increase productivity and better user experience to the IDE. Let's explore them one by one.

Previewing images in Solution Explorer

Solution Explorer shows a preview of images just by hovering on the image without actually opening the image. This is a new enhancement to **Solution Explorer** and is not available with any previous version of Visual Studio IDE.

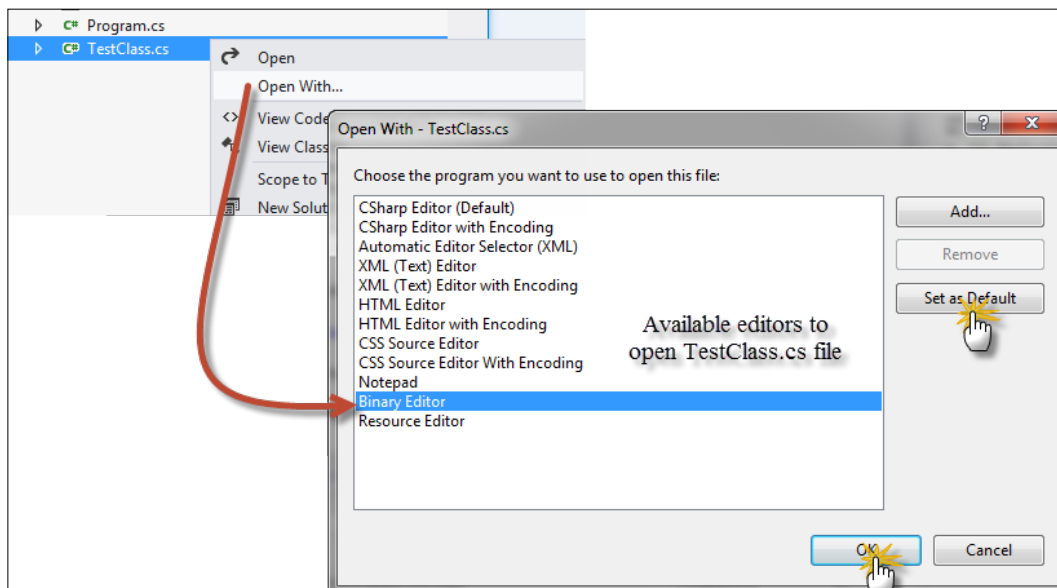


Add an image in the solution by right-clicking on the project and navigating to **Add | Add Existing Item** and select an image from the filesystem.

If you did it correctly, the image will be loaded in the tree as in the screenshot. Just hover the mouse pointer over the image, and you will see a small preview of the image as shown in the previous screenshot.

Different IDE editors

Visual Studio comes with a number of editors installed within it by default. Based on the type of the file, this editor gets loaded onto the IDE. For instance, if you double-click on an image it will open it in image editor, while when you choose a `.cs` file, it will open it in a C# editor.



You can right-click on any file from the `Solution` and select **Open With...** rather than using the normal double-click to open a dialog box, which lists all the available editors that can load the selected file. Some of the default IDE editors are C# Editor, C# Editor with Encoding, Automatic Editor Selector, XML Editor, HTML Editor, Notepad, Binary Editor, Resource Editor, and so on.

To open a CS file in Binary Editor, right-click on the CS file and choose **Open With...** choose **Binary Editor** and select **OK**. You can see from the following screenshot that the code file looks like a sequence of binary characters:

00000010	0D 0A 75 73 69 6E 67 20 53 79 73 74 65 6D 2E 43	...using System.C
00000020	6F 6C 6C 65 63 74 69 6F 6E 73 2E 47 65 6E 65 72	ollections.Gener
00000030	69 63 3B 0D 0A 75 73 69 6E 67 20 53 79 73 74 65	ic...using Syste
Address	Hex Value	Text Equivalent

The first column shows the address of the bytes in the file, the second shows the actual byte content, and the third contains the string equivalent of the same.

You can also try out other editors in the list.

Working with the main workspace area of IDE

This section represents the main workspace area of the screen. This is the most important section of the Visual Studio IDE which the developers mostly use. The workspace generally fills up the entire IDE or most of the portion of the IDE. Each of the windows that can be loaded inside the IDE has the feature to toggle hidden, can float outside the IDE, and even be snapped into different dock positions.

In the main workspace area, a file has already been loaded for us with a class named `class1`. The editor associated with `.cs` file is loaded in the screen to show the file. There are a number of editors available with Visual Studio, each of them can be loaded directly in this section. Generally, we do our main development in this section of the IDE.

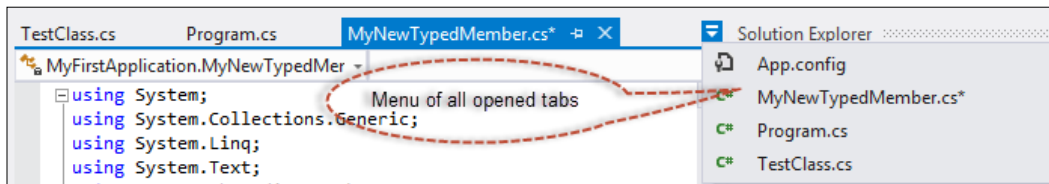
How to do it...

We will work the main workspace area of IDE by performing the following steps:

1. Close all associated windows in the IDE that you can see the portion of the IDE that still remain on the screen is the main workspace area.
2. Open **Solution Explorer**, double-click on some files. You can see each file produces a stack of tabs. Upon opening a new file, you can see the new tabs are stacked on the left-hand side of the tabs. Opening a large number of files in the IDE will produce a menu on the top-right corner of the screen.
3. Drag a tab and place it in between other tabs to reposition.
4. Change something in the file without saving the content. The tab header will indicate the update with a star sign. It will show a lock sign when you open a read-only file.
5. Use the **Toggle** button on one of the tabs to make it sticky, so that opening new files does not changes its position. If you are in the **Preview** tab, you will see a special **Promote** button, which will promote it as a new window to work on. The workspace contains the editor which forms the most of the part of IDE. This section loads the content of the actual file. The changes in the editor are tracked in yellow (when the change is not saved) and green (when the content is saved).
6. You can zoom the content of the editor using the **Zoom** dropdown in the bottom-left corner of the screen.

How it works...

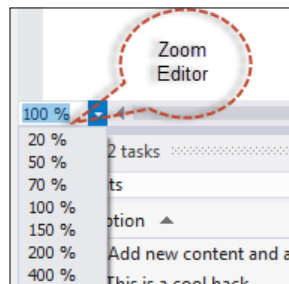
The workspace loads the editors in tabs. So, when you pick two nodes from **Solution Explorer** to open the code window, Visual Studio keeps links to each of the files that are opened in separate tabs. Each tab header contains a few fixed set of items.



In the previous screenshot, you can see that the tab header containing the name of the file (`MyNewTypedMember.cs`) that links to the tab, it shows a * when the item needs to be saved, it has a toggle pinner button (just like all other IDE tool windows), which makes the tab sticky on the left-hand side, and the close button. The title section sometimes also indicates an additional status, like when the file is locked it shows a lock icon, when the object is loaded from metadata it shows that in square braces as in the screenshot. In this section, as we keep on opening files it goes in to a stack of tab pages-one after another until it reaches the end. After the whole area is occupied, it finally creates a menu in the right most corner of the workspace title to hold a list of all the files that cannot be shown on the screen. You can select from this menu to choose which file you need to open. *Ctrl + Tab* can also be used to toggle between the tabs that are already loaded in the workspace.

Below the title of the tab, before the main workable area, there are two dropdowns. One has been loaded with the class that is opened in the IDE and the right one loads all the members that are created in the file. These dropdowns help easier navigation in the file by listing all the classes that are loaded in the current file on the left. On the right-hand side there is another which contextually lists all the members that are there in the class that is chosen on the right-hand side. These two dropdowns are smart enough to update automatically whenever any new code is added to the editor.

The main workspace area is bounded by two scroll bars that handle the overflow of the document. But after the vertical scroll bar, there is a special button to split the window.



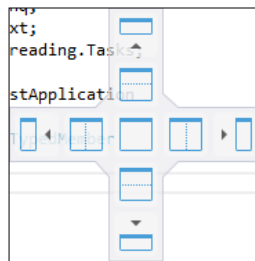
The horizontal scroll bar on the other hand holds another dropdown that shows the current zoom of the editor. Visual Studio now allows you to scale your editor to your preferred zoom level. The shortcut for the zoom is *Ctrl* + the scroll mouse wheel.

There's more...

As our basic overview of the Visual Studio 2012 IDE is over, let us have an insight on a few other features that are available inside the IDE.

Docking windows inside the IDE workspace

Let's go on opening a few of the windows in the IDE. You can start from **View | Windows Menu** of the IDE. After you have opened up quite a number of tool windows, you will find a requirement for easy arrangement of the windows. Visual Studio IDE is composed of a number of dock holders, each of which can freely flow both inside or outside of the main IDE. You can move a window by dragging its title bar as you do for any normal window. When you move the panel just inside Visual Studio, a set of controls appear as shown in the following screenshot, which indicates the dock positions available for the current window:



When you hold against the dock position, it will show you how it appears when stacked in a certain position (shown in the previous screenshot). Visual Studio will automatically adjust the stacks of items that are stacked through the Dock Managers.

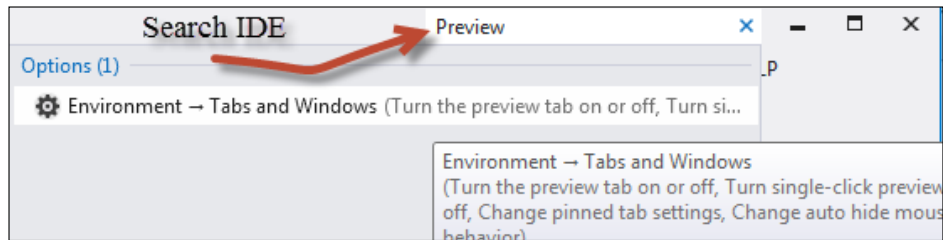
The whole IDE has a stack of tabs on each side of its workspace, which holds tabs to link each of these windows. As I have already mentioned, each of the tabbed window is capable of toggling to auto hide. When the window is hidden it shows only the tab link, while when the window is open, it associates itself into a set of tabs.

The other parts of the IDE consists of the menu bar, the standard toolbar, and the status bar. Each of them is used to give commands to the IDE.

Search IDE features

On the top-right corner of the screen, you will find a new search box. This is called the IDE search box. Visual Studio IDE is vast. There are thousands of options available inside IDE, which you can configure. Sometimes, it is hard to find a specific option that you want. The IDE search feature helps you find an option more easily.

As shown in the previous recipe, say if I forget where the option for **Preview File** tab on single click is available, I can type `Preview` in the search box.

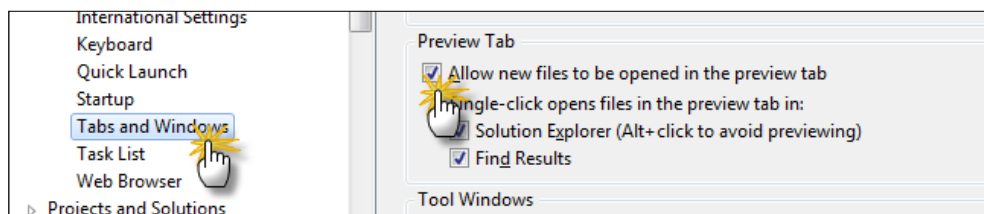


As shown in the screenshot, Visual Studio will open a list of all the options where you can see the `Preview` text available. On choosing the first option, you will navigate to the appropriate location.

Preview files in IDE

Visual Studio IDE has a feature to preview files without opening the same in the IDE. Just with a single click on the file, it will open up in the main workspace area as preview. As we have already seen, the preview of the file is generally opened on the right-hand side of the IDE, and selecting another file replaces the previous preview, it is called as a temporary preview of file. The file can be promoted or opened either by working directly inside the preview, or by using a special promote button on the title of the preview tab.

Generally this option is disabled by default. You can navigate to **Tools | Options**, then in the left tree, **Environment | Tabs and Windows**, check the **Allow new files to be opened in preview Tab**.



The **Preview Tab** option is generally useful when you are working with a large number of files and loaded in the IDE.

Navigating between code inside the IDE

Visual Studio comes with a number of useful code navigators; most of them are pretty useful and handy. First of all let us try the **Code Highlighting** feature in Visual Studio.

How to do it...

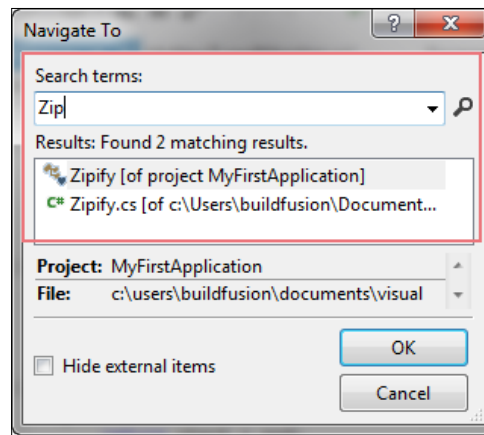
To try the Code Highlighting feature in Visual Studio, perform the following:

1. Double-click on a C# file to highlight each occurrence of the same type in the code.
2. Press *Ctrl + Shift + down/up* arrow to navigate between references.
3. Press *Ctrl + ,*(comma) to open the **Navigate to** dialog box to list all the navigation options available for the current selection (or under cursor position).
4. Right-click on any method or type and select **Go To Definition** to move to the definition of it.
5. Right-click on any of the type and select **Find All References** to list all the references in a new tool window.
6. You can select **View Call Hierarchy** from the right-click pop-up menu to list all the references to and from the member you have selected.
7. You can rename a type in the IDE and press *Ctrl + .*(dot) to open a menu and rename all its references.

How it works...

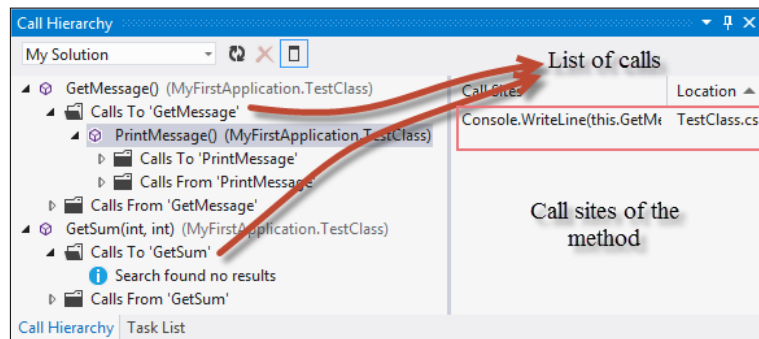
Navigating from one file to another and one type to another is an important thing while working with any project. Visual Studio IDE did a splendid job by giving us tools and features that help in navigating from one place to another easily.

Select any text in the project document by double-clicking on it, which highlights all the occurrences of the same code in the file. You can press *Ctrl + Shift + down/up* arrows to navigate between the references:



For advanced code navigation, you can press **Ctrl + ,** to open a new window named **Navigate To** that quickly searches code members and files and list them.

The **Navigate To** dialog box also shows the file from which the window is invoked and the name of the project.



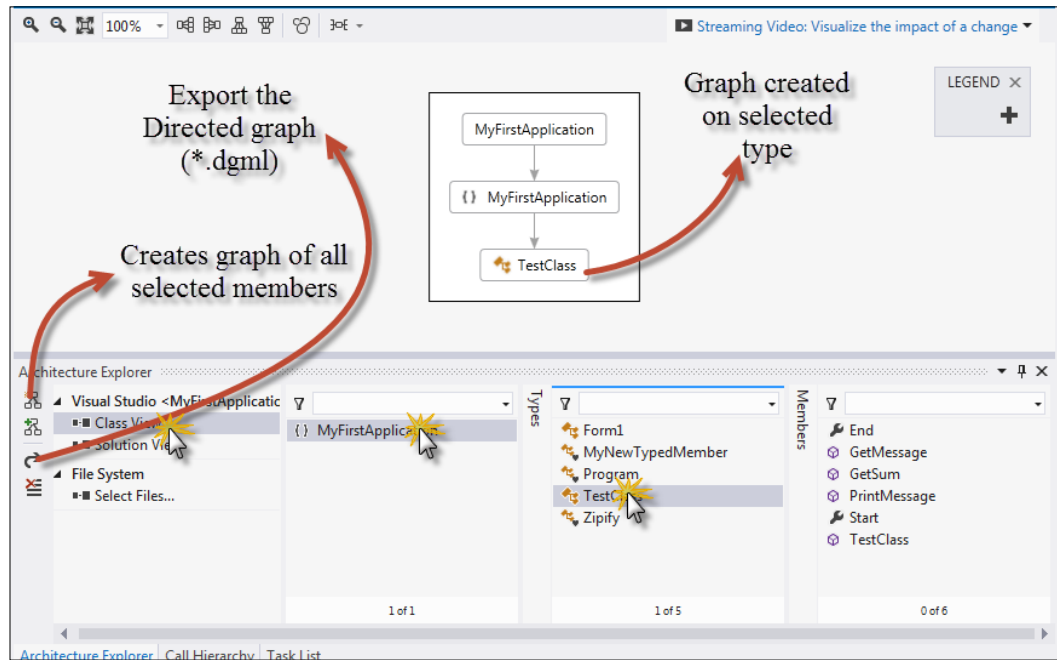
Another important code navigation tool is the **Call Hierarchy** window. You can invoke this window using right-click and selecting **View Call Hierarchy**. The **Call Hierarchy** window gives you the overview of all the references that are using the same code. You can also view the overrides of the method if you have any. The right-hand side of the window also lists all the calls by giving the exact line and the filename where the call has been made.

There's more...

In addition to the basic navigation tools inside the IDE, there are a few advanced options available to the IDE that help in navigating within the UI quickly, logically, and efficiently. Let's now take a look at a few other options available to us.

Architecture Explorer

One of the coolest additions to Visual Studio recently is **Architecture Explorer**. The tool helps you to navigate between the solutions assets very easily. Let's navigate to **View | Architecture Explorer** to get a window similar to the following screenshot:

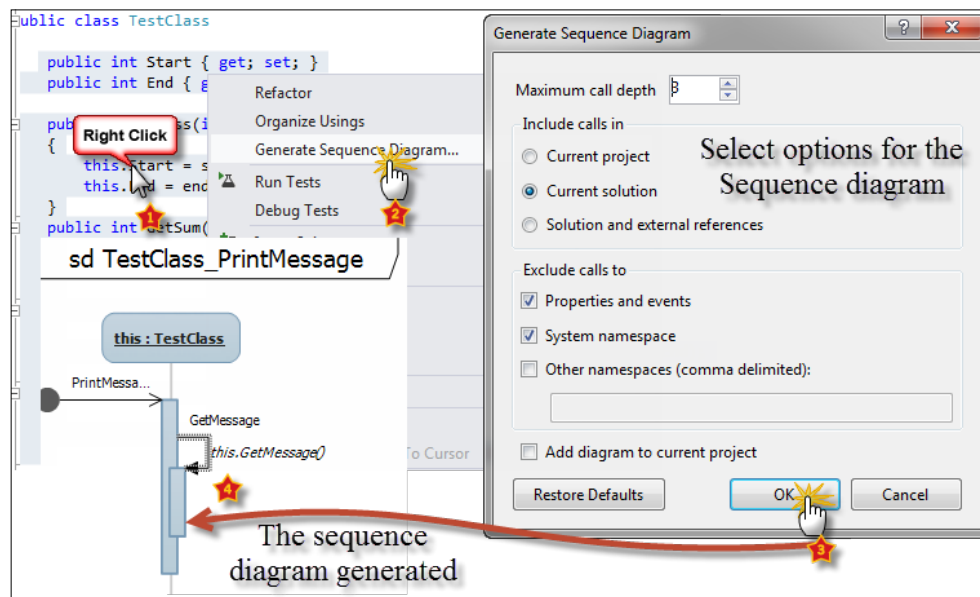


In the **Architecture Explorer** window you can view, navigate to the content of any class, namespace, or method. Visual Studio also has the facility to export the **Architecture Explorer** window in a graph document. You can select the items that you need to show in the graph and select the **Create New Graph Document** button on the top-left corner of the **Architecture Explorer** window. The graph will show up to analyze code members, circular references, unreferenced nodes, and so on and create a pictorial representation of the entire library. You can even export the document as a directed graph using the **Export Directed graph** button on the explorer.

The graph in the **Architecture Explorer** window can be exported to XPS document for future reference too.

Sequence diagrams

You can generate sequence diagrams from Visual Studio IDE by directly right-clicking on a method and by selecting the **Generate Sequence Diagram** option. This option is only available in the Ultimate version of Visual Studio. **Sequence diagram** will show a diagrammatic representation of the complete method body using a diagram.



You can see in this method that I have used a few classes and objects, which are shown in the diagram based on their usage.

Task List

Visual Studio can be used to list task information on the project. You can open **Task List** from the **View** menu, which lists all the tasks that are outstanding in the project.

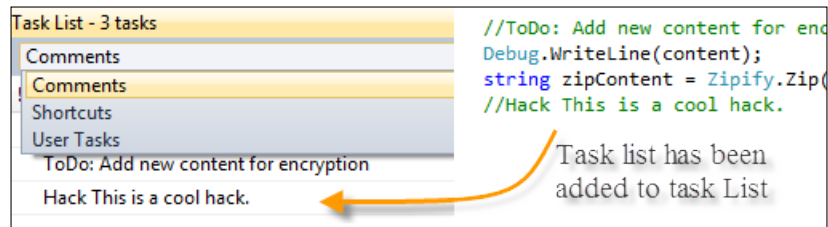
There are a number of options that you can use to create tasks in the project.

- When you comment something in the code with **Task List** tokens, your task will get listed in the **Comments** section. For example:

```
// ToDo This is a task that is outstanding or
// Hack This hack need to be changed
```

The **ToDo** task will be listed on the **Tasklist Tool** window.

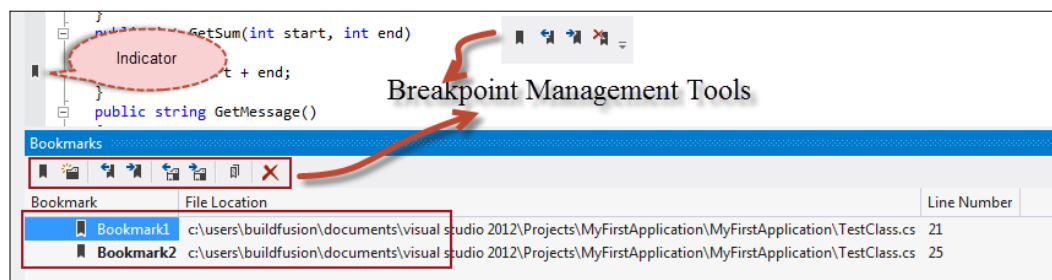
- ▶ You can also create tasks directly inside the task window. Choose **User Tasks** and click on the button just beside the combo. You can also specify the priority of a task.



The commented tasks can be double-clicked to navigate to the appropriate line where the comment is written.

Bookmark menu

Another important window to manage code is **Bookmarks**. You can also use Bookmarks to navigate between code. To add a bookmark, go to the line where you want to apply a bookmark to, and select the **Toggle Bookmark** option in **Edit | Bookmarks**, or press **Ctrl + B, T**. A white box will appear against the line. Once the bookmark is set, you can move to next and previous bookmarks using **Ctrl + B, N** and **Ctrl + B, P** respectively. You can clear all breakpoints either from the menu or simply choosing **Ctrl + B, C** from the keyboard. You can also open the **Bookmark** tool window to navigate between bookmarks more easily.



The **Bookmark** tool window can be opened using **Ctrl + W, B** or by navigating to **View | Other Window | Bookmarks**. You can manipulate bookmarks from this window.

The Code Definition window

This is a read-only editor present inside the IDE which displays the definition of types and methods while the user navigates on the code in the editor. As you move the cursor over the IDE or change the selection on Class View, Object Browser, or Call Browser, the content of the **Code Definition** window automatically gets updated with either the actual code from within the application, or it displays metadata content of a selected type.

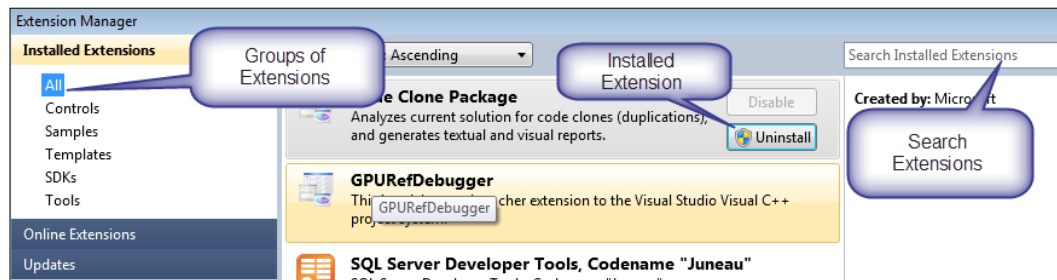
To open the **Code Definition** window, navigate to **View | Code Definition** window.

The **Code Definition** window not only displays the definition of the code in the navigation, but it also allows you to copy code, use **Edit Definition** to edit the definition, put breakpoints, and so on. The **Code Definition** window is very useful in certain cases while working with large applications.

Extension Manager

Visual Studio supports extensibility. A large number of Visual Studio IDE components are now extensible. **Extension Manager** is a special section which allows you to view, control, or uninstall any extension associated with Visual Studio.

Navigate to **Tools | Extension Manager** to open the window. If you choose **Online Extension** from the left-hand side, it will connect to the online extension gallery. You can select an extension from the list and download the extension and install.



Once the extension is installed, it will show you the option to either disable or uninstall the extension.

What is MSBuild and how can I use it?

The Microsoft Build Engine is a platform to simplify the build process when there are a large number of files that need to be compiled together. Visual Studio build process uses the MSBuild environment to provide transparent build experience on all the files and folder structures together in one library. The entire build process needs a project file, which is an XML-based file that provides the basic structure of the library.

Visual studio has a project file to maintain all the items that are included in the project and this file is later passed on to the MSBuild interface to invoke build process. In case of advanced scenarios when you don't have Visual Studio available, but you need to build a hierarchy of project structure, MSBuild can be invoked manually too by writing the project file manually.

Let us examine each section of a project file.

- ▶ **Item:** It represents the items that comprise the build process. They are grouped together into user-defined collection.

```
<ItemGroup>
    <Compile Include = "Program.cs" />
    <Compile Include = "TestClass.cs" />
</ItemGroup>
```
- ▶ **Properties:** It presents the key/value pair of all the properties that configure builds.

```
<PropertyGroup>
    <BuildDir>Build</Build>
</PropertyGroup>
```
- ▶ **Task:** They are tasks that needs to be performed while the build process is on.

```
<Target Name="MakeBuildDirectory">
    <MakeDir Directories="$(BuildDir)" />
</Target>
```
- ▶ **Targets:** They form the entry point of the build process. Targets are grouped into individual build process.

```
<Target Name="Compile">
    <Csc Sources="@ (Compile) " />
</Target>
```

These sections provide valuable information about how the project needs to be built. After successfully creating the project file, you can use it to create the actual executable:

```
MSBuild.exe TestApplication.proj /property:Configurtion=Release
```

Thus, the project will be built in the release mode.

Debugging the application

After the application has been created successfully, the next step is to run the application from inside the IDE and debug it. Debugging an application inside Visual Studio IDE is fun. You can execute the application step-by-step to clearly understand the code that is running and also identify any problem(s) that the code might have while executing.

To debug the application, either you click on **Start** from the toolbar, or select **Debug | Start Debugging** from the menu. The shortcut for start debugging the current application is *F5*.

When the application runs inside the IDE, every step the application performs is monitored by the IDE, and any changes made to the application directly pass through the IDE execution host engine. Breakpoints are special indicators inside the code which allows the IDE to halt the execution of the program at a certain point. When the application breaks at a breakpoint, the program stops its execution and everything in the state gets evaluated in the environment.

To go over the program line by line, we can either choose **Debug | Step over** or press *F10* in the IDE, or to step into the definition of the code, we can navigate to **Debug | Step into** or press *F11* from the IDE. Visual Studio provides a new environment for the IDE to execute the program while debugging. The environment also supports a large number of tools that help in clearly identifying what is going on at a particular point for the application.

See also

You can try the productivity power tools for Visual Studio 2012 for other extensions to the IDE available at <http://bit.ly/ProductivityPowerTools>.

Extending Visual Studio templates

Visual Studio files are created using templates. There are a large number of templates associated with Visual Studio that are stored with your IDE, which are automatically copied when you create files and projects in the application. These files are called **templates** in Visual Studio.

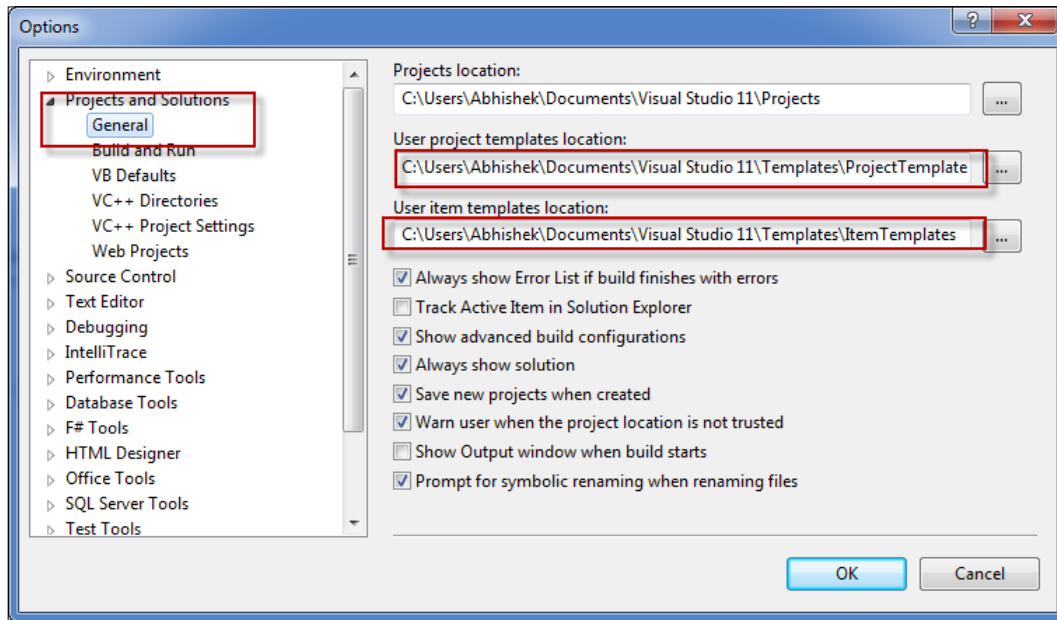
We can use Visual Studio to produce custom templates that can automate these things for us, by using the pre-defined structure for the project file type such that whenever you start a new project or add a new item to your project, most of these items are generated for you.

In this recipe, we will cover how to use all the predefined templates that are preinstalled with in the Visual Studio IDE and also talk on how to create a template of your own.

There are two types of templates:

- ▶ **Project Templates:** These files are related to projects and are used when a new project is created or added to a solution. The templates that we create in the project are listed inside the **New Project** dialog box of Visual Studio.
- ▶ **Item Template:** These are item files that are listed in **New Item** dialog box of the project. So when we add a new item to the project, the item templates that are listed in the dialog box are deployed as **Item Templates**.

The default location of project and item template folders is listed in the **Options** dialog box of Visual Studio as shown in the following screenshot:



In the previous screenshot, you can see that the Project and Item templates are located to a specified unc path. They are basically a zipped content archive that we can place in the same location to ensure that it is listed on the project and item dialogs.

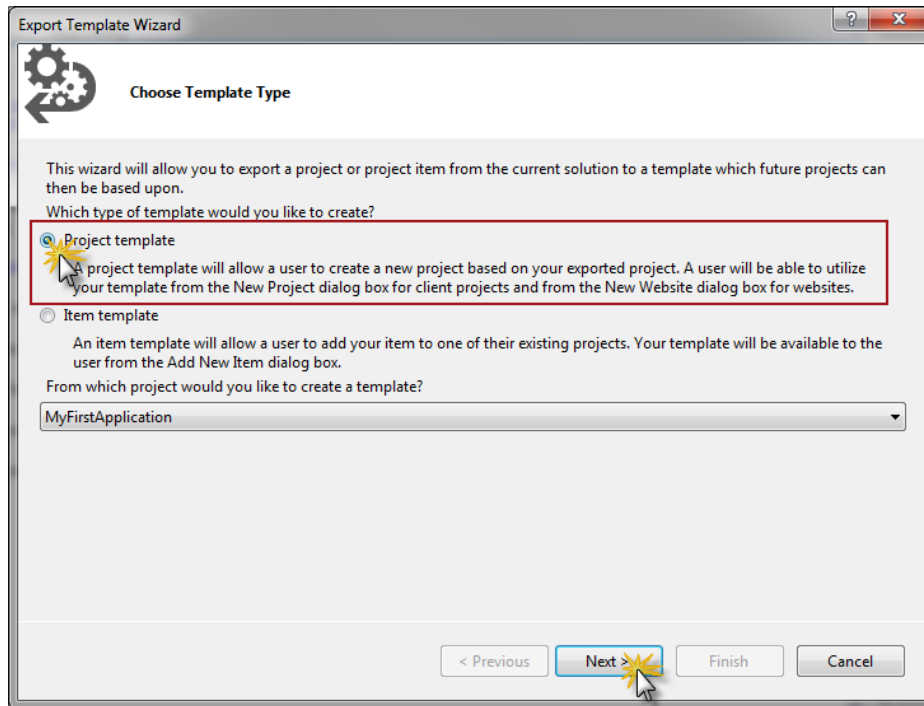
The basic **Project Templates** and **Item Templates** are located inside the Program Files\ [Microsoft Visual Studio Installation Directory] \Common7\IDE\Project Templates and Program Files\ [Microsoft Visual Studio Installation Directory] \Common7\IDE\ItemTemplates folders respectively.

How to do it...

Let us create a new project and item templates in this recipe and look into their details.

1. Create a custom project by writing the code that is actually needed whenever someone uses the template.
2. Include references of assemblies that need to be referenced on the final project.

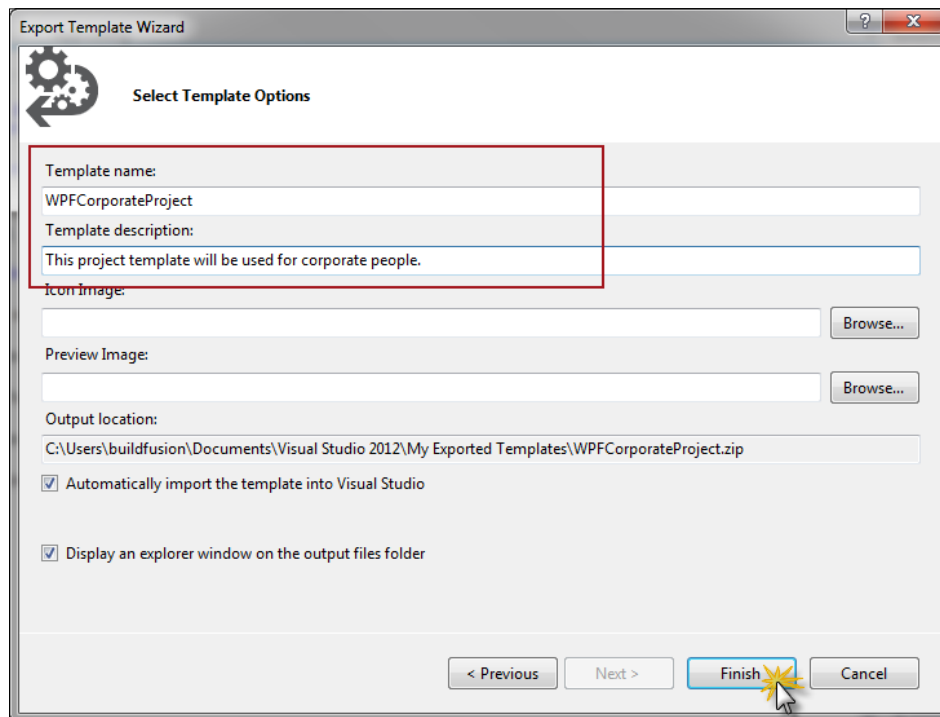
3. Select **File | Export Template**. A new wizard will open up where you can choose either **Project Template** or **Item Template**. Let us choose **Project Template** so that the entire project gets exported.



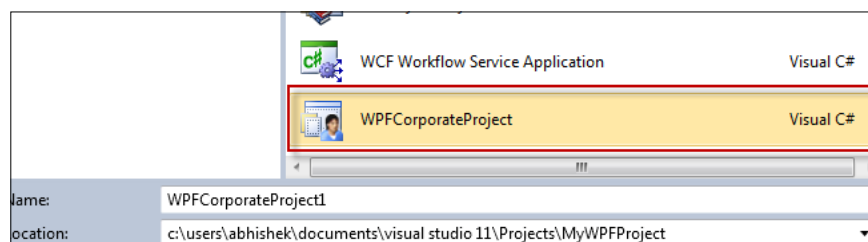
If you have multiple projects in the same solution, you can use either of them from the dropdown at the bottom of the screen.

4. In the next screen, we need to specify the name of the template which it needs to show in the **New Project** dialog box and description of the template. This information will be shown inside the **New Project Template** dialog box either in tooltip or in the separate description pane.
5. The **Template Options** window also gives you options to choose the icon file that is to be used and a preview image for the template icon. In our case, we leave it blank.

6. Notice, the output location is shown in a textbox below these options and it shows that a ZIP file will be created in the location specified:



7. Once everything is done, click on **Finish**. The `WPFCorporateProject.zip` file will be created. The file is actually exported to `My Exported Template` location. To make it list in the **New Project** dialog box, you need to copy the file to the template location.
8. Finally when we open the **New Project** dialog box, it shows our **WPFCorporateProject** as shown in the following screenshot:



9. When we select the project type, it will automatically replace the namespace name, filenames, and so on, and customize itself for the new project.

In the same way, you can export **Item Template** in the solution. The main difference between the **Project Template** and the **Item Template** is that the Project Template is meant to define the whole project, while the Item template is meant to define one single project item.

How it works...

Visual Studio defines a set of rules that need to be applied to the content of a ZIP file, when we create a new instance of it. During the export of a template, Visual Studio analyzes all the files that need to be exported with the project such that it can recreate the file again. It zips the whole content to a compressed zipped archive and exports the same to the specified folder.

Let's look at the content of the zipped archive.

The main file that holds reference to all the information about the template is inside the `.vstemplate` file. It is an XML file with the name of the project, the description, the project type, the default location, icon, and the references to the project items are listed in the file. You can look into the file for details:

```
<VSTemplate Version="3.0.0" xmlns="http://schemas.microsoft.com/
developer/vstemplate/2005" Type="Project">
  <TemplateData>
    <Name>WPFCorporateProject</Name>
    <Description>This project template will be used for corporate
people</Description>
    <ProjectType>Windows</ProjectType>
    <ProjectSubType>CSharp</ProjectSubType>
    <SortOrder>5</SortOrder>
    <CreateNewFolder>true</CreateNewFolder>
    <DefaultName>WPFCorporateProject</DefaultName>
    <ProvideDefaultName>true</ProvideDefaultName>
    <LocationField>Enabled</LocationField>
    <EnableLocationBrowseButton>true</EnableLocationBrowseButton>
    <Icon>__TemplateIcon.ico</Icon>
  </TemplateData>
  <TemplateContent>
    <Project TargetFileName="MyWPFProject.csproj" File="MyWPFProject.
csproj" ReplaceParameters="true">
      <ProjectItem ReplaceParameters="true"
TargetFileName="App.config">App.config</ProjectItem>
      ...
    <Folder Name="Properties" TargetFolderName="Properties">
      <ProjectItem ReplaceParameters="true" TargetFileName="AssemblyInfo.
cs">AssemblyInfo.cs</ProjectItem>
      ...
    </Folder>
  </TemplateContent>
</VSTemplate>
```

```
</Folder>
</Project>
<References>
  <Reference>
    <Assembly>System, Version=4.0.0.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089
    </Assembly>
  </Reference>
  <Reference>
    <Assembly>
      System.Data, Version=2.0.0.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089
    </Assembly>
  </Reference>
</References>

</TemplateContent>
</VSTemplate>
```

In this sample XML document you can see there are few things to address:

- ▶ The Name and Description indicate what needs to be shown on the New **Project** and **New Item** dialog boxes to the user. The Type and description appear on the description pane of the dialog box.
- ▶ ProjectType and SubType indicates the grouping information of the Project Type in the **New Project** dialog box.
- ▶ SortOrder defines where the item will be placed by default.
- ▶ LocationField indicates whether the location of the project could be chosen or not. By location we mean the unc path of the hard disk location.
- ▶ TemplateContent identifies the whole hierarchy of the structure of the exported Template. These may contain references, project files, folders, and so on.
- ▶ Each project contains a set of ProjectItem. ReplaceParameters indicates a set of routines that needs to be run on the file to ensure that all the parameters that are assigned to the file are replaced with the actual value. For instance the parameter \$safeprojectname\$ is replaced with the actual project name that we specify while creating the project.
- ▶ A project can also include folders. The Folder element can be used to enumerate all the files inside the Folder for a specific project.
- ▶ You can add some custom references to .dlls using Reference that are available for each project. The references can be multiple.

When we open the **Project Template** dialog box, it reads the XML file with the .vstemplate extension and lists the template accordingly.



It is recommended to use reference from GAC rather than actually ship the .dll inside the zipped archive.

The template that has been created using the wizard is pretty simple and straight forward. Generally in real world scenarios, it is not very useful with the basic template that is there in Visual Studio. We need to export the template to include additional components. When Template is created we generally need to replace certain parameters that need to be set with actual values when the template is used up. Let's look into what are the parameters that are already predefined to be replaced by Visual Studio while creating the actual file.

Parameter Name	Description
clrversion	Replaces the current version of CLR.
GUID [1-10]	You can generate 10 GUIDs for a single project. The parameter will be replaced with the unique GUIDs in the files.
itemname	Name that the user provides in the Add Item pop-up dialog.
machinename	Name of the computer.
projectname	Replaces the project name that the user enters into the Project Popup dialog box.
registeredorganization	Organization Name to which the computer is registered. It takes the value from the HKLM\Software\Microsoft\Windows NT\CurrentVersion\RegisteredOrganization registry key.
rootnamespace	Namespace to add project files into. It returns the current root Namespace information.
safeitemname	Returns the current File name.
safeprojectname	Returns the project name chosen by the user in the Project Dialog.
time	Returns time of file creation.
userdomain	User domain name.
username	Fetches current user name.
year	Year of file creation.

We need to add a \$ sign before the start of each of these parameters to use it in the file. Let us have a look at a sample code with parameters that we need to use while editing the projects.

```
// Legal Notice goes here
// File Created : $year $time$
// Created By : $username$ $userdomain$ $clrversion$
```

```
// Copyright protected : $registeredorganization$

using System;
using System.Collections;
using System.IO;
using System.Text;
using MyOrganization.$safeprojectname$;

namespace MyOrganization.$safeprojectname$
{
    public class $safeitemname$
    {
        #region Variables
        //Declare Variables
        #endregion

        # region Properties
        // Declare Properties
        #endregion

        # region Events
        //Declare Events
        # endregion

        #region Constructors
        //Declare Constructors
        public $safeitemname$()
        {
        }
        #endregion

        #region Methods
        //Declare Methods
        #endregion
    }
}
```

The parameters in the file will get replaced when the actual file is created. Just add a new item from the list and you can see the parameters are replaced with meaningful names. Try out yourself and see the magic.

There's more...

After creating a template for your project, let us add some sugar to the plate by introducing some of the additional points that you might need to consider while using this in real-world scenarios.

Adding custom parameters

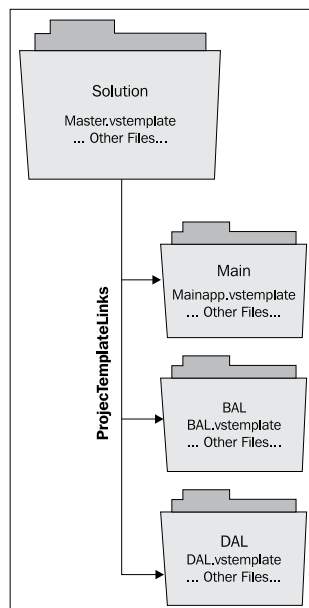
Sometimes, the parameters that are available to the files do not seem enough to you. Visual Studio templates allow you to put custom parameters, which will also be replaced when the actual replacement occurs. To write your custom parameters, you can use the `CustomParameters` tag and list a collection of all of those parameters that will be replaced while creating the item. Lets see how it works.

```
<TemplateContent>
  <CustomParameters>
    <CustomParameter
      Name="$mycustomparameter$"
      Value="Template designed by @abhishek" />
  </CustomParameters>
</TemplateContent>
```

Inside `TemplateContent` of the `.vstemplate` file, we can write our custom parameters like `mycustomparameter`. which will be replaced by the value.

Creating a template with more than one project

Instead of using one single project as an item of project, it is sometimes useful to create an entire solution as a template. For example, for every organization the development group generally not only starts a project using one single project at a time, but rather they will create extra projects for data access layers, business logic, and so on inside the solution in addition to the actual project. We can create templates for all of them and list all in the **New Project** dialog box, or creating a template with the entire solution and giving it to all the developers is an easier way.



Multiple projects can be created from one ZIP file, but it needs to include one master `.vstemplate` file which lists the links to all individual projects in the solution. Let us suppose we want to have three projects for the template, one for Data layer, one for Business Logic, and another one is the actual project. To do this, we need to individually create all the `.vstemplate` files as we have discussed in the steps.

Once you have created all of these templates, we create individual folders for each of the projects. Data Access Layer should reside in DAL, Business Logic will reside in BAL, and the actual project will remain inside the Main folder.

The main content of the `SolutionTemplate` will look like the following:

```
<VSTemplate Version="2.0.0" xmlns=http://schemas.microsoft.com/
developer/vstemplate/2005
  Type="ProjectGroup">
  <TemplateData>
    ...
  </TemplateData>
  <TemplateContent>
    <ProjectCollection>
      <ProjectTemplateLink Projectname="DataAccessLayer">
        .\DAL\DAL.vstemplate
      </ProjectTemplateLink>
      <ProjectTemplateLink ProjectName="BusinessLogic">
        .\BAL\BAL.vstemplate
      </ProjectTemplateLink>
      <ProjectTemplateLink ProjectName="MainApplication">
        .\Main\MainApp.vstemplate
      </ProjectTemplateLink>
    </ProjectCollection>
  </TemplateContent>
</VSTemplate>
```

You can easily notice that the type of the template hence defined is actually a `Project Group` rather than a `Project`. Here we have defined the `ProjectTemplateLink` attributes to different projects templates in the zipped archive. You should create separate folders called BAL, DAL, and Main to store the `.vstemplates` of each individual project and place the `SolutionTemplate.vstemplate` file in the parent folder.

Once you are done with it, you can zip the entire folder structure again and put it in the configured location to be loaded on the **New Project** dialog box.

See also

- ▶ <http://bit.ly/VSTemplates>

Using Code Snippets in Visual Studio

Code snippets are another important feature of the Visual Studio IDE. Visual Studio IntelliSense menu lists a number of code snippets that could be used very easily when you code for fast development. Code snippets are also available in the right-click menu in the editor as **Insert Snippets**.

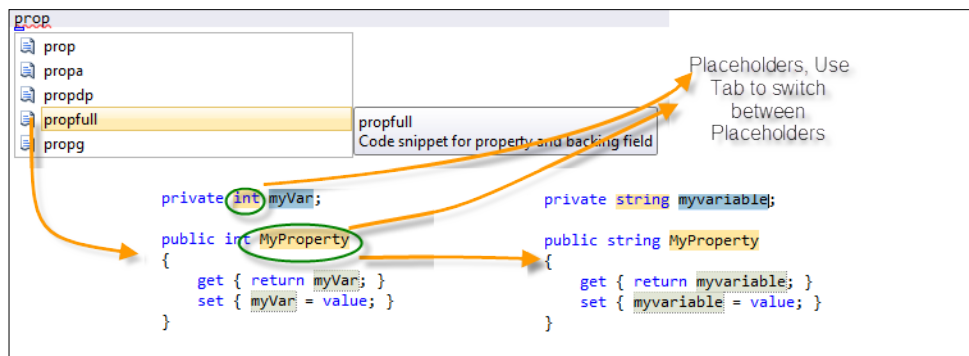


IntelliSense is a feature that enables writing code faster and easier. It is a feature that auto-completes the code while writing it in the IDE. Just start typing, it will automatically get the associated contextual items and list them inside the code window. For writing code faster, IntelliSense comes very handy.

How to do it...

To insert code snippets, perform the following steps:

1. Inside a class, write `prop` and press `Tab`.
2. The property Code snippet will appear on the screen. The `propfull` tab will look as shown in the following screenshot:



3. A code snippet is composed of a number of placeholders. You can toggle between these placeholders using the `Tab` key and type the appropriate code.
4. Snippet is smart enough to automatically change placeholders that has a common meaning. In our case, if you change the data type of the variable, it automatically changes the data type of the property. You can change the name of the variable, and it will change all other occurrences of the variable in the code snippets.
5. When you are happy with the snippet, you can press `Enter` to continue.

You can also insert code snippets by right-clicking on the IDE and choosing **Insert Code snippets**. This option gives you the entire hierarchy of code snippets from which you need to choose appropriate code snippets.

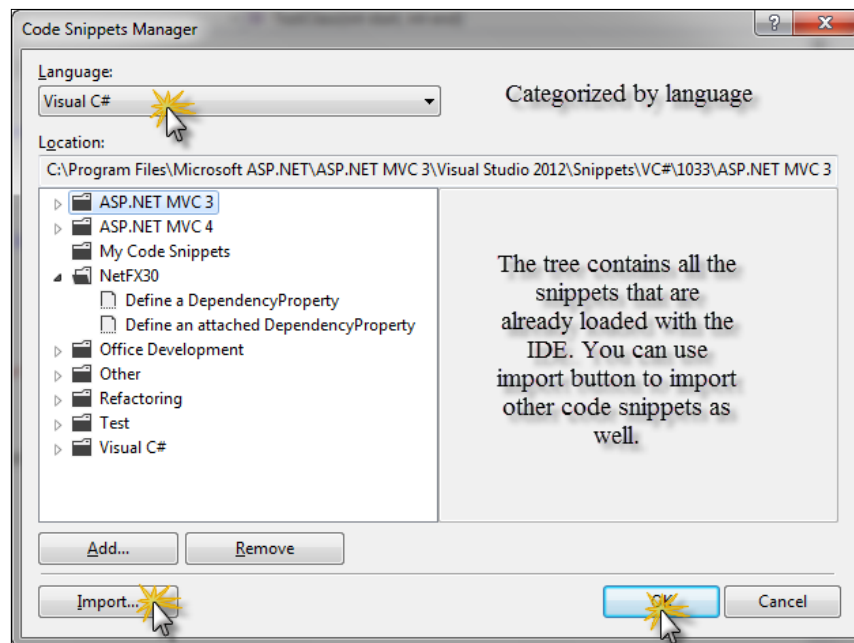
Code snippets are available virtually in every type of code. You can use code snippet for ASP.NET pages, C# code blocks, or even for a JavaScript element.



There is a special type of code snippet available in Visual Studio called Surround with. The Surround with code snippets are items that produce outlining on the code. For instance, you can Surround with a code with an if block which supports writing code inside it, or even you can surround a combination of methods inside a region. You can select a code block and right-click to select the Surround with feature of the IDE.

How it works...

Visual Studio IDE also comes with an in-built **Code Snippets Manager** to easily manage code snippets. To open it, navigate to **Tools** and Choose **Code Snippets Manager**.



In the previous screenshot, you can see you can easily manage the property/code snippets within **Code Snippets Manager**. You can get the location where the actual code snippets are installed. You can even import a new code snippet to your Visual Studio IDE. By default, the custom VS code snippets are loaded from `MyDocuments/Visual Studio/Code Snippets`. You can create your own code snippet and store it into this folder, so that it will be automatically detected and loaded in the IntelliSense menu.

Code snippets are actually an XML file with specific schema. Generally most of the code snippets that are regularly used are already listed there in the IntelliSense menu.

But while developing an application, it is often required to use `INotifyPropertyChanged` for properties. Let's see the following code which will automatically include `INotifyPropertyChanged` for a property:

```
<?xml version="1.0" encoding="utf-8" ?>
<CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/
CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>propnch</Title>
      <Shortcut>propnch</Shortcut>
      <Description>Code snippet for property and backing
field and ensure that it invokes INotifyPropertyChanigng and
INotifyPropertyChanged</Description>
      <Author>Abhishek</Author>
      <SnippetTypes>
        <SnippetType>Expansion</SnippetType>
      </SnippetTypes>
    </Header>
    <Snippet>
      <Declarations>
        <Literal>
          <ID>type</ID>
          <ToolTip>Property type</ToolTip>
          <Default>int</Default>
        </Literal>
        <Literal>
          <ID>property</ID>
          <ToolTip>Property name</ToolTip>
          <Default>MyProperty</Default>
        </Literal>
        <Literal>
          <ID>field</ID>
          <ToolTip>The variable backing this property</ToolTip>
          <Default>myVar</Default>
        </Literal>
      </Declarations>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>
```

```
        </Literal>
    </Declarations>
    <Code Language="csharp"><![CDATA[
private $type$ $field$;
public $type$ $property$
{
    get { return $field$; }
    set
    {
        this.OnPropertyChanging("$property$");
        $field$ = value;
        this.OnPropertyChanged("$property$");
    }
}
    $end$]]>
    </Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>
```

The code here is divided into three sections. The `<Header>` section includes the `<Title>`, `<Description>`, `<Shortcut>`, `<SnippetType>`, and so on. These information are read and shown from the IntelliSense menu.

The second section represents the actual body of the snippets that is divided into the `<Declaration>` part, and the `<Code>` block. The `<Declaration>` part defines the `<Literal>` attributes that are to be used in the `<Code>` block which will be replaced actually by the user. The `<Literal>` attributes are the placeholders of the current block.

The `<Code>` block includes the actual code. Here we define the `<Literal>` attributes inside `$`. Save the file inside the default location, that is, `My Documents\Visual Studio 2012\Code Snippets`. Once you install the Code snippet, it will be available inside the IntelliSense menu.

There's more...

There are additional features to the IDE that can also help in the productivity of the code. Options like removing unused using statements and collapse/expand outlines can really help in navigating within the code more efficiently and productively. Let us consider some of the other interesting features within the IDE that can help productivity.

Organizing Usings

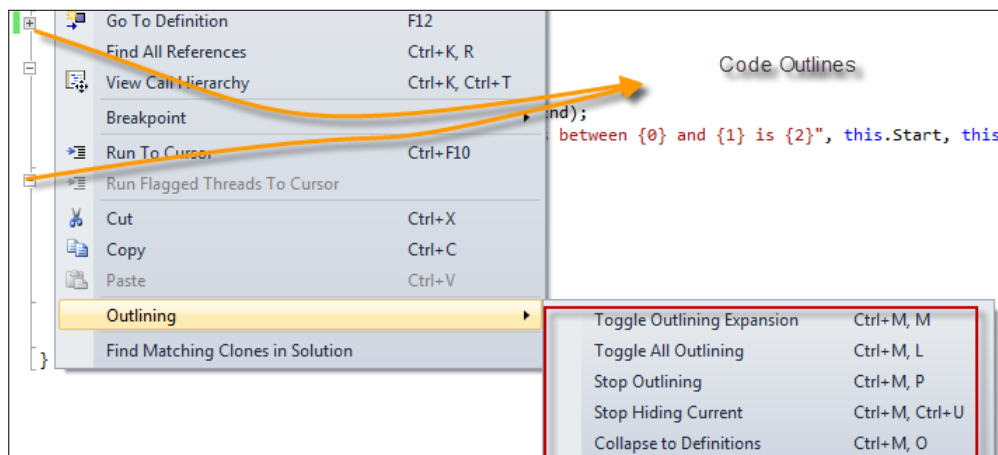
Visual Studio also allows you to organize Usings on your code by removing unnecessary Usings or reordering using statements at the top. It analyzes and parses the code that you wrote on the file upon selecting the option and does these manipulations.

To use this, right-click on your code and select **Organize Usings**. There are a few options available under this:

- ▶ **Remove Unused Usings:** This will check every type that you have used in the code inside appropriate namespaces and remove those which are not used in the code.
- ▶ **Sort Usings:** This option will sort the Usings in alphabetical order.
- ▶ **Remove & Sort Usings:** It is the combination of both.

Outlining

Visual Studio is smart in analyzing the code. When your code is loaded into the IDE, it is parsed and loaded with outlines, which allow you to hide/unhide individual blocks in the code. It produces a collapsible region of code which produces a + (plus) sign when collapsed and - (minus) when expanded.



Collapsing the outline shows only the portion of the code that is relevant to identify it. Hence, while you are writing code in a large file, it is nice to have all the blocks collapsed. For a large code, it is a very tedious job to collapse/expand all outlining. The Visual Studio **Outlining** menu lets you toggle between them easily.

Right-click on the code and use **Outlining | Collapse to Definitions**. This will collapse all the logical blocks of code recursively into their collapsed state.

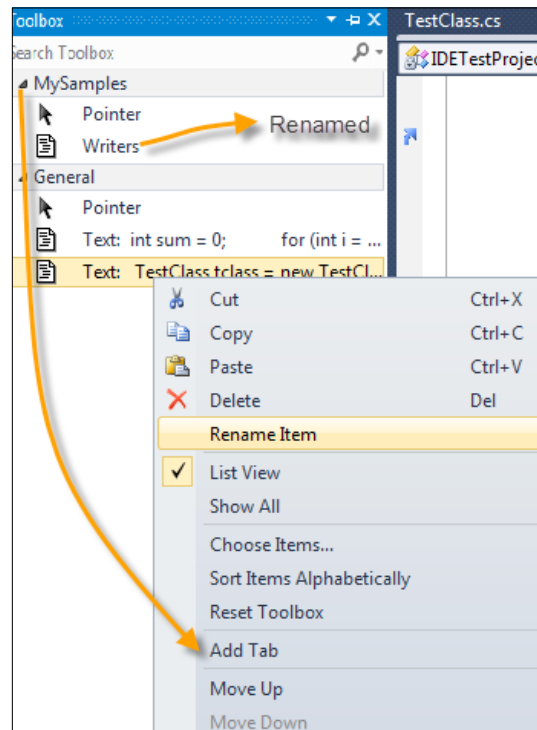
You can use **Toggle All Outlining** to toggle between expanded mode and collapsed mode.



You can also disable the **Outlining** feature of code using the **Stop outlining** feature from the menu. Once the outlining is disabled, you can re-enable it using **Start automatic outlining** again.

Using the Toolbox


Visual Studio has a Toolbox which can help you in some cases to quickly find code that you require often during development. Generally, the Toolbox is useful to work with the Designers, but sometimes it comes very handy to increase your productivity by remembering the recurring code.



Select a code that you need very often. Drag it from the editor to the toolbar on the left-hand side of the IDE. If it is not open, navigate to **View | Toolbox** from the menu or press **Ctrl + W, T**.

The Toolbox will produce an item automatically with **Text:....** You can hover over the toolbar item to see what code it has.

To use the code, you can again drag it to the editor.

 You can also rename or delete a code snippet by right-clicking on the item in the toolbox. You can even click on **Add Tab** to group common code blocks in one place.

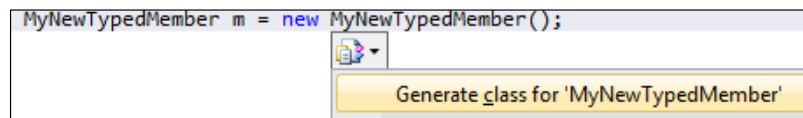
Using Smart Tags and Refactor in Visual Studio

Visual Studio allows you to use **Smart Tags** to enhance the productivity of writing code. Visual Studio Smart Tags can produce blank implementation of properties, methods, fields, or can even create a custom type by creating a class file directly in the appropriate project.

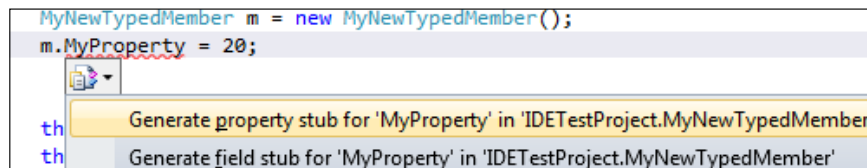
How to do it...

In this recipe, we are going to cover how to use the Smart Tags and Refactor option inside Visual Studio 2012.

1. Start typing the code as shown in the following screenshot. You can see while writing that Visual Studio is smart enough to detect the class name in the IntelliSense menu even though the Type does not exist.



2. By invoking `Ctrl + .` (dot) on the screen, it will open a small smart menu which allows you to create a Type just for the statement. Notice that the Type will create a file in the solution and add it to the project as well.
3. Let's create some members inside the Type. Go on assigning members, press `Ctrl + .` (dot) to produce properties inside the file as shown in the following screenshot:



The Smart Tags can also be used for various other productivity needs like renaming a Type, renaming a method, implementing an Interface, and so on. Let's have fun with creating few more Types and Methods in the solution. Sometimes this approach is also called as **Consume First Deployment**.



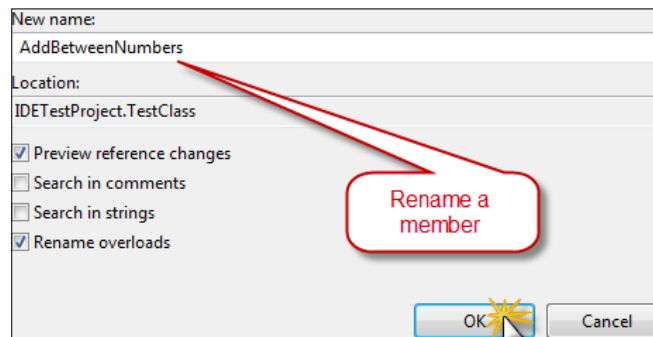
Similar to Smart Tags, Refactoring is another important feature of the IDE which helps in refactoring of code blocks easily. Refactoring is actually a technique to change the software system in such a way that the internal structure of the design changes are made more intuitive while the external behavior of the system remains the same. The **Refactor** Menu adds a special meaning when you are working on large projects. It is taken as a rule of thumb that when writing a code, you should always consider a method to only contain one unit of work and the whole Type or class should contain one independent module of work.

4. Consider the following code block:

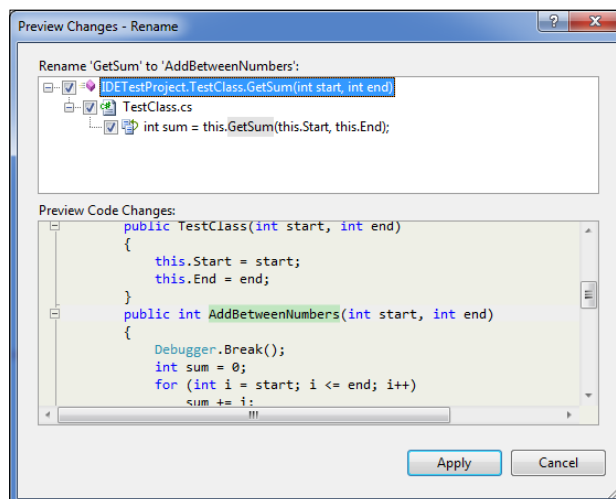
```
public void PrintMessage()
{
    int sum = 0;
    for (int i = this.Start; i <= this.End; i++)
        sum += i;
    string message = string.Format("Sum of all numbers between
{0} and {1} is {2}", this.Start, this.End, sum);
    Console.WriteLine(message);
    Console.ReadKey(true);
}
```

5. The above method is doing more than one unit of work in a single method. It computes the sum of all the numbers between start and end, and then finally prints out the message. Let's select the first three lines and choose **Extract Method** from the **Refactor** menu. **Extract Method** is capable of automatically analyzing the code and detects the appropriate parameter sets that needed to be passed from your code and the appropriate return statement.
6. A new window pops up to allow you to export the selection to a new method. Let's call it `GetSum`.
7. Click on **OK** to finish. The magic of Refactoring automatically changes the portion of code with a method call and the method is placed in the code just below it.
8. Let us suppose that at a certain point of time, we don't like the name of the member name `GetSum`, and we need to rename the member name more appropriately and want this to be done to all overloads and every calls that are made to in the entire project, all at a time. One thing you can do is to use the Find option in Visual Studio. But this will be worse when two types have the same method name. This has been a common problem. The Refactor Rename feature gives you easy means to rename a method by automatically analyzing code based on its type used. Select the member you want to rename and Press *F2*, or select `Rename` from the Refactor Menu. Say we want to rename the method `GetSum` to `AddBetweenNumbers`.

9. Select the **New Name** in the dialog box that comes up, and select the options that are needed to be considered.




- ❑ The **Preview reference changes** will indicate that the tool will show a new dialog box that will be previewed before changes are made. Always select this option to be on the safe side.
 - ❑ **Search in comments** indicate that the name in comments also need be changed.
 - ❑ **Search in strings** will search in string literals too.
 - ❑ **Rename overloads** are useful if we need to change name of all overloads of the current name.
10. Once we select all the options, we go on to do the actual Rename operation. Now as we have selected Preview reference changes, the Refactor in Visual Studio will go on and select all the method names on each Type associated with it to change names and will be presented in a new dialog box in a tree format.




As you can see in the previous dialog box, it searched one occurrence of `GetSum` in the tree `TestClass.cs`. It also previews the code on what it will look like after conversion is been made in the bottom pane. You can individually choose the checkboxes from the tree and make changes everywhere you need. The Nodes are responsive, I mean you can preview any node from the tree view upon clicking on the type and the actual final code will be shown on the pane below.

11. Finally when everything is OK, select **Apply** to apply changes to every node that is checked.

[ For quick rename, you can also use **Rename Smart Tags** that renames all references automatically.]

Sometimes, it is important to encapsulate a Field into a property. One way is to write the entire property yourself, or you can navigate to **Refactor | Extract Field** to do the same thing. **Extract Field** works the same way as **Rename**. Upon selecting the field of a class, you will be asked for the name of the property and you can use this to create a property from the field.

12. **Extract Interface** creates an interface from the existing type taking all the members from the current type and exporting it into a new interface, and finally implementing the class from the interface.
 1. Create a class that has at least one instance specific data member.
 2. Right-click on the class and navigate to **Refactor | Extract Interface**.
 3. You will be prompted with a dialog box where you can specify the name of the interface, the list of members in the current type and the filename.
 4. Click on **OK**.

[ If you already have an interface and want to auto create methods, you can right-click on the interface and select **Implement Interface** or use **Smart Tags**.]

13. **RemoveParameter** or **ReorderParameter** are other options in the **Refactor** menu to remove extra parameters or to reorder the parameters in a member. Sometimes when the parameter is long, these options come in handy. Select each of these options and try them inside the IDE.

How it works...

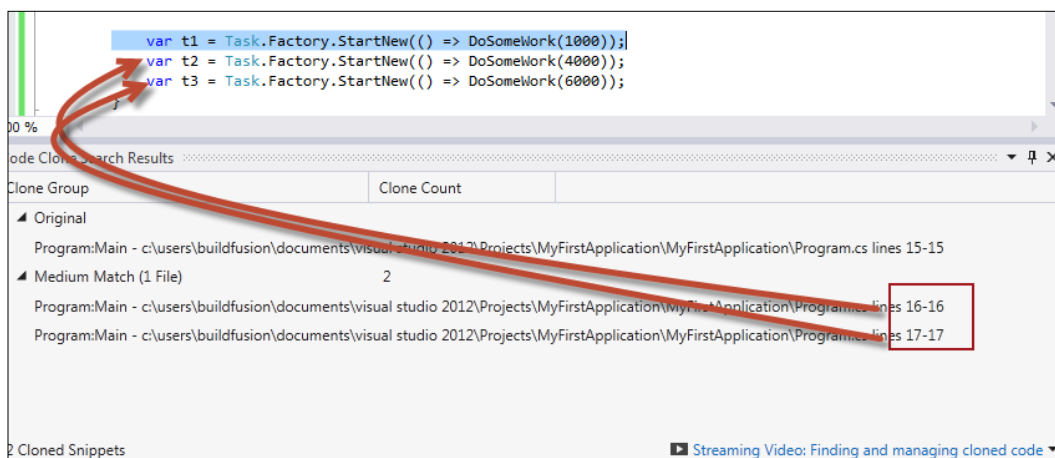
Smart Tags and Refactoring are inbuilt to the Visual Studio editor and act as a productivity feature to the developer that automatically creates objects or refactor them. There are a large number of these available within the IDE that can be used while coding to increase productivity.

There's more...

Even though Refactoring and Smart Tags are the most important parts of code analysis, Visual Studio IDE also contains some other interesting features that are worth noticing. Let us consider them here.

Clone Code Detection

Visual Studio IDE includes a unique feature to detect clones in code, or in other words you can find duplicate code in your solution. Sometimes developers copy and paste an existing code to other part of the project. Visual Studio Code analyzer finds code that is almost similar in logic. It is capable of detecting variable changes and parameters. To use this feature, select the portion of code that you want to find clones for. Right-click and choose **Find matching clones in solution**. You can also select **Analyze Solution for Code Clones** to detect any clones in the solution.



While the IDE detects the clones, it automatically puts in the weight of the clone.

Sometimes it is required to exclude files from the result of Clone search. This is often required when you have generated similar code using tools. In such cases, you can place a `.codeclonesettings` file inside the main application directory with the list of files that do not include in this search. This file is actually an XML file containing files, namespace, types, and so on which are excluded from the search. For instance:

```
<CodeCloneSettings>
  <Exclusions>
    <!-- Absolute or relative path names: -->
    <File>TestFile.cs</File>
    <!-- Filepaths may contain wildcards: -->
    <File>GeneratedFiles\*.cs</File>
    <!-- Namespace, Type, and FunctionName must be fully qualified:
    -->
    <Namespace>MyCompany.MyProject</Namespace>
    <Type>MyCompany.MyProject.MyClass1</Type>
    <FunctionName>MyCompany.MyProject.MyClass2.MyMethod</FunctionName>
    <!-- Names may contain wildcards: -->
    <Namespace>*.AnotherProject</Namespace>
    <Type>*.AnotherClass*</Type>
    <FunctionName>MyProject.*.AnotherMethod</FunctionName>
  </Exclusions>
</CodeCloneSettings>
```

The wide variety of wild cards makes this very handy to create logic.

See also

- ▶ <http://bit.ly/VSSmartTags>
- ▶ <http://bit.ly/VSRefactor>

2

Basics of .NET Programs and Memory Management

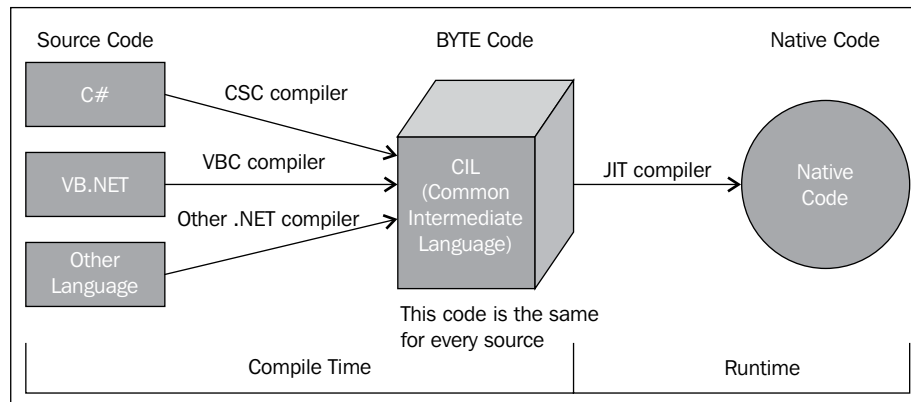
The goal of this chapter is to get you through the introduction of a .NET program, its files, and core components. The idea behind this chapter is to give you an insight on how .NET is built and we will be closely looking at the memory usage of a .NET program. In this chapter we will cover:

- ▶ Inspecting the internal infrastructure of a .NET assembly
- ▶ Working with different types of assemblies
- ▶ Inspecting the major components of a .NET program
- ▶ How to work with custom configuration for an application
- ▶ How to disassemble an assembly
- ▶ Securing your code from reverse engineering by using obfuscation
- ▶ Knowing the .NET garbage collection and memory management
- ▶ How to find memory leaks in a .NET program
- ▶ Solutions to 10 common mistakes made by developers while writing code

Introduction

.NET has made a lot of improvements in its core to enhance the performance of an application. It is the first solid-core environment that really connects to all parts of the technology for developers. After getting through with the basic understanding of the Visual Studio tool in the previous chapter, you would be definitely waiting to start writing code in the IDE. But to make solid footsteps in the .NET environment, it would be great to know about the basic advantages and also the architecture of the .NET environment that you are going to start working on. In this chapter we will mostly cover the basic infrastructure of a .NET environment to give you a solid foundation.

The most important infrastructural component of the .NET framework is **Common Language Runtime (CLR)**. The code that you write inside the framework is called the **managed code**, which benefits from cross-language integration, cross-language exception handling, enhanced security, versioning, and easy deployment support. Every component in a managed environment conveys some specific rule sets. When you write your code in a managed environment, the code gets compiled into the **common intermediate language (CIL)**. This language remains constant or equivalent to a code that might be compiled from another language. The commonality of languages is maintained in a managed environment by the use of an intermediate language. The languages that are built on top of CIL follows rules defined on the **Common Language Specification (CLS)**. Let us demonstrate the fact from the following diagram:



Here in the diagram, the source code from any .NET language is converted to CIL code with help from the respective supported compilers. To compile a C# source code, we have CSC compiler and similarly we have a VBC compiler for VB .NET code. The CIL is then again converted to native code using the JIT compiler. The JIT (just-in-time) compiler converts the intermediate language to a machine-dependent code when it is executed during runtime. Thus, you can copy an assembly to any platform that has the runtime installed and the assembly will work in the same way. The introduction of a multicore JIT implementation with CLR makes applications run almost at the same pace as running native code.

The CLR is supported by **Common Type System (CTS)**. When you consider the languages in .NET, each of them is backed up with a wide set of libraries available in every language. This library is a set of assemblies installed when the .NET framework is loaded in a system, and when it forms a major portion of any program, we built on top of it. Even though the framework allows you to write code in different languages, the internal type system remains constant for every language. For example, `Integer` of VB .NET is mapped to the same type in CLR as to `int` of C#. Both of them point to the `System.Int32` type in a framework.

The language that exists in the .NET environment is built on top of CLR such that the compiler that compiles the source code should produce the CIL which follows the ECMA standardization specified. There are a number of languages that are not built by Microsoft and are not shipped with the .NET framework which include **Lsharp, Boo, A Sharp, Fantom**, and so on, which are built on top of the .NET framework and which follows the ECMA standards. The internal bits of an assembly built by these language compilers are exactly same and can be identified by the JIT compiler easily.

Inspecting the internal structure of a .NET assembly

In .NET when you compile your code, an **assembly** is produced. An assembly is a collection of modules (or even a single module) which holds the intermediate code and metadata associated with it which are popularly known as **Portable Executable** files or in short **PE** files. Each assembly in .NET can be linked together with other assemblies via an **assembly linker**. We cannot see the data inside a .NET assembly using text editors. We need specialized tools that can read binary data present inside an assembly to get any meaning from it. The .NET framework includes a great tool called `ildasm.exe` that can read from portable executables and show you the IL. The idea behind inspecting an assembly is to understand what is inside it and get to know some of the concepts that are worth knowing about the files that we will work on. Knowing the bits and pieces can be of great help in understanding the behavior of a .NET assembly on different runtime environments.

Getting ready

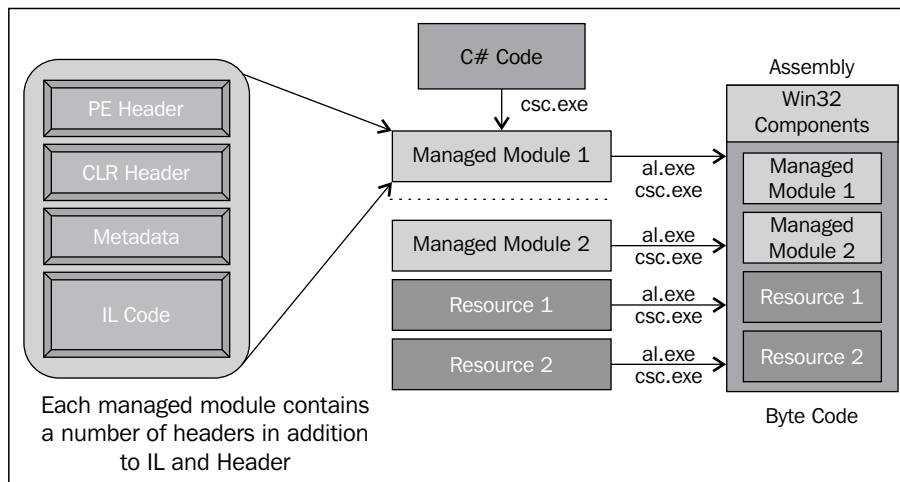
Before we start the actual recipe, we need to create an assembly so that we can examine it further. An assembly can be either a .dll file (dynamic link libraries) or an executable which can be produced from a project. So let's open Visual Studio and create a **Console Application** project. Type in the following code:

```
static void Main(string[] args)
{
    Console.WriteLine("Inside Main");
    AnotherMethod();
}

private static void AnotherMethod()
{
    Console.WriteLine("Inside AnotherMethod");
}
```

The preceding code is very simple. It has one main method block which prints something on the screen and another method called `AnotherMethod` that prints another message. Now if you run this program, it produces these messages onto the screen.

When you go to the `Debug\bin` directory of the project, you will see that the actual executable has been created there. We are going to use this assembly to examine its contents:



In the preceding pictorial representation, we depicted how a managed module looks and how the resources are written inside the actual assembly component. Each managed module contains a number of headers and its metadata in addition to the actual IL instructions which form the program logic. Once each managed module is built, it is then combined to form an assembly using assembly linker. Each managed module is contextually found and loaded using a relative virtual address by CLR. Let us look into the actual content of a module in this recipe.

How to do it...

1. Open **Visual Studio Command Prompt** and move to the location where the executable is located. Copy the path of the location, type `CD` and right-click to paste the path so that the command looks like `cd "your path"`. Press the return key (*Enter*) to move your current directory to the specific path. For example, write the following command:

```
dumpbin /all "yourassemblyname.exe" >output.txt
```



`dumpbin` is a utility program from Microsoft that reads the information about a portable executable file and produces it as a text content.

This will produce a file on the same directory called `output.txt` which contains information about the assembly.

2. Open `Output.txt` to see the output hence generated. You will find that the file is divided into different parts.
3. The first part of the section identifies the PE header section of the assembly. It defines the file type, file header values, and optional header values. Each of these sections together comprise the assembly. Each section starts with a section header, which determines the header information about the specific header.
4. Moving down the optional parameters, you will see the information about stack reserve and committed size and heap reserve size. These options define what will be the reserved stack size for all the threads running on the process or the allocation of heap in the process. The greater this size the less the allocation needed for the application. But also know that the overall size of the memory used by the process increases considerably.

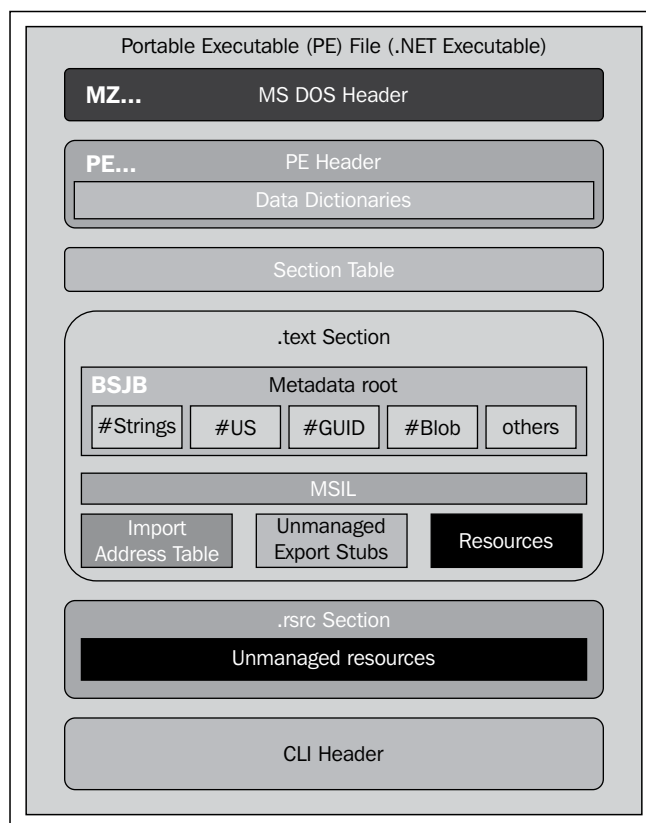


Note that these options will only be found when you are looking at an executable file.

5. If you scroll down further, you will see a number of important directories. In our case, we see that it lists 10 directories which are represented by pointers using **Relative Virtual Address (RVA)**. These addresses are specified as an offset from the base address where the PE file has been loaded into memory.
6. As in the previous diagram, you can see there are a number of directories defined. The most important one is the COM descriptor directory which describes where exactly the metadata and IL start in the assembly.
7. Scrolling further it shows the actual sections. In our case there are three sections available which are sorted by the starting address (RVA). The first one represents the metadata information about the assembly. Each of the raw metadata information has been marked as an offset. You can see the number corresponding to each line representing the address of the line.
8. The Debug directories are also listed in the file with the absolute path of the .pdb file. A unique GUID has also been created for the build which matches the correct .pdb file.
9. The CLR header represents a special meaning to the compiler. You can see in the diagram the CLR header has an entry point token assigned to 600001. This is the number from where the execution starts. The CLR header also lists the runtime version.
10. Another interesting thing to address; the start metadata definition block is marked with 4 bytes, 42 53 4A 42 (BSJB), which are the first letters of the names of the developers who are responsible to the implementation of metadata section in .NET. **BSJB** stands for Brian Harry, Susan Radke – Sproull, Jason Zander, and Bill Evans.

How it works...

Assemblies are code contained in Microsoft Portable Executable files with metadata and all other associated information inside. An assembly can be made up of more than one file but that one file should contain metadata information about all other files in a section called **manifest**. Let's take a look at the different sections of an assembly file.



Like any executable, every .NET assembly starts with **MZ** header information (coined after the initial of **Mark Zbikowski**, who has been the original architect of MS-DOS). This has been kept just to print out the message **This program cannot be run in DOS mode**.

After the MS-DOS Header information, the PE Header information starts. When you see the output file of the `dumpbin` utility, you will see the PE header section, starting with some of the basic information regarding the managed module with some data directory information. The data directory represents special information inside the file which stores a predefined set of values. The location of each directory is marked with a pointer address to RVA.

The real starting point of a PE file, from the .NET point of view, is the COR20 header. This tells the .NET runtime where to find the metadata. The COR20 header specifies some data directories just like the PE header as well as the entry point of the program. The address of the start of metadata streams is pointed in one of these directories.

.NET holds the metadata information in a number of streams each of which is in a different format:

- ▶ **#BLOB**: This holds binary data, which includes method signatures, strings in UCS 2.
- ▶ **#US**: This stores strings in UCS 2.
- ▶ **#GUID**: This stores the streams of GUIDs that are used in the assembly. GUIDs are referenced in the stream using indexes not an offset.
- ▶ **#String**: These streams comprise of most of the part of the assembly which includes information about UTF 8 strings which include names of types, methods, and so on.
- ▶ **#~Stream**: This stores metadata table information.

Finally the MSIL which represents the intermediate language is stored. The MSIL forms the major portion of the assembly which needs to be JIT compiled when the assembly is executed.

The metadata section also lists a separate directory for the import address table and unmanaged export stubs and resources, although the unmanaged resources have a separate section (`.rsrc`). Each of these directories is mapped by the CLR header information.

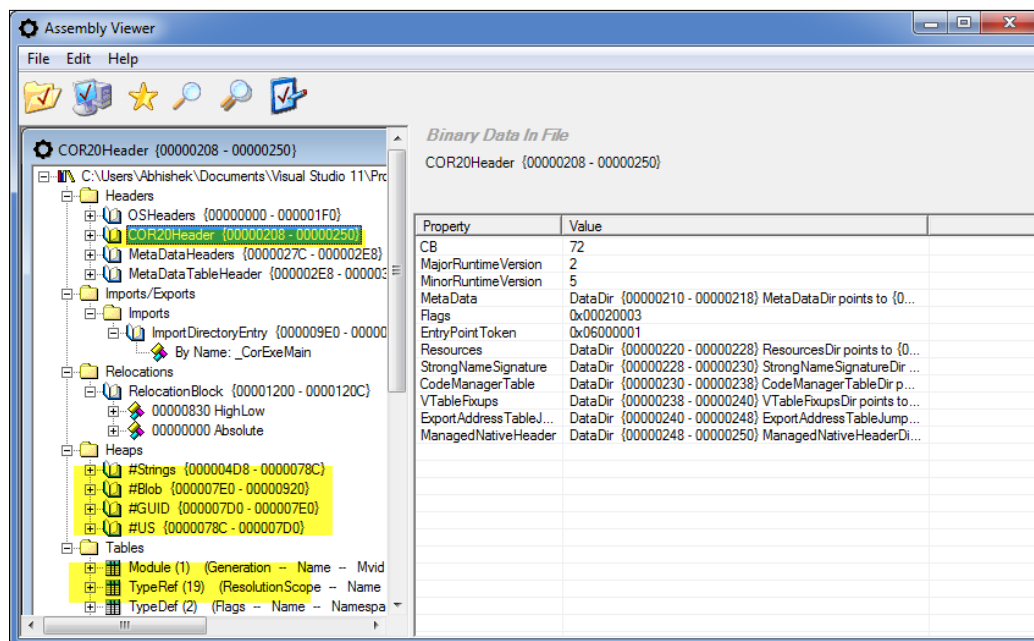
There's more...

There are also a number of other options used to inspect an assembly. Let's look into some of the other ways of inspecting an assembly.

Inspecting by using an assembly viewer

Asmex created by Ben Peterson (<http://bit.ly/AssemblyViewer>) is an amazing tool to inspect the inside of an assembly. Please run the tool from the source code included with the chapter that would help us inspect an assembly.

There are a few options that you need to consider when using the tool. The first one is to open a private assembly. Let's run the tool and open an assembly as shown in the following screenshot:



When you open an assembly in this tool, you will see a tree on the left-hand side which is divided into sections such as **Headers**, **Imports/Exports**, **Relocations**, **Heaps**, **Tables**, and many more as shown in the preceding screenshot. Please consider looking at each section one by one.

The **Headers** section includes, **OSHeaders** which includes the information that is required for every binary file in the Windows operating system, **COR20Header** which includes metadata information of CLR runtime, **MetaDataHeaders** which includes the location of streams, storage signatures, and so on, and **MetaDataTableHeaders** which includes information regarding metadata tables. These headers are marked clearly using the offset location for that assembly, such that, if you open the assembly and read to an offset to that address, you will find exactly the same information that is included in the tool.

Now going further to **Heaps**, you will see the streams of storage that have been written into the assembly. These streams are separated into blob, GUIDs, Strings, and UCS streams. If you select these nodes, you can see what is written into the assembly.

You can also consider looking at **Tables**, which gives you the information on all the bits and pieces of the assembly. There are a number of nodes that appear for tables, such as **TypeDef**, which has the information about all the types that exist inside the assembly; or **Constants** that write the information about all the constants defined within the assembly.

If you try to open an assembly from **Global Assembly Cache (GAC)** using the **Open Assembly** button of this tool, you can see additional information about the assembly as well. For instance, you will see what referenced assemblies are for a particular assembly, or the exact file location for the assembly, and so on. If you want to learn more, it is recommended that you look at each section of the assembly using this tool and discover more hidden secrets of an assembly.

Working with different types of assemblies

In the previous recipe we have understood the internal bits of an assembly. We saw the components that comprise an assembly; the file formats and metadata information regarding an assembly. We have built an executable and inspected its major components. The main motive of the previous recipe was to understand different sections of a .NET assembly and its file format. In this recipe, our focus is to build an assembly with different components.

A .NET assembly is of three types:

- ▶ **Private assemblies:** Private assemblies are those files that are deployed with the application, either in the same directory where the application is running or in its subdirectories. These assemblies are designed to be used by one particular application and to remain private to all other applications. To enable another process to use a private assembly, it needs to be deployed in the same location where the DLL is deployed. Each of the processes that share a private assembly, loads the assembly into its own domain. Private assemblies do not use a strong signature to identify the file when it is called for. They load a disk file directly into the memory where the process is allocated.
- ▶ **Public assemblies:** They are installed in a separate location called **Global Assembly Cache (GAC)**. A public assembly is shared between applications and has version constraints. In .NET, more than one assembly with the same name may exist when they are from different versions. A public assembly is required when either you are working on a very large project that requires generalized modules or the component belongs to third-party vendors.
- ▶ **Satellite assemblies:** They are .NET assemblies that store language-specific resources. You can place different resources from different languages into different assemblies and the correct assembly will be loaded when application is launched.

If you consider the assembly based on file types, they are of two types. The first one consists of executables that can run independently, while the other consists of DLLs which are associated to other executables to run its components.

Getting ready

Let's start Visual Studio and create a **Console Application** project. In the **Solution Explorer** pane, right-click and select **Add New Project** and select **Class Library**. We will use class libraries to demonstrate each of the assembly types one by one.

How to do it...

After creating a new project, let's try each of the types of assembly one by one by carrying out the following steps:

1. When Visual Studio is ready, write the following code inside `Class1.cs` (I have renamed the file to `MyClass.cs`):

```
public class MyClass
{
    public void AMethodOnaClass()
    {
        Console.WriteLine("Code inside a Method");
    }
}
```
2. Now take reference of the library to the console application. To do this, we right-click on the project and select **Add Reference**, and from this dialog box, we choose the project from **Project Reference**, and compile.
3. After this if you move to the `bin` directory of the console application, you will see that the `PrivateAssembly` is present in the same location.
4. Let's add another **Class Library** template to the project and now let's call it `SharedAssembly`, and write the following code:

```
public class MysharedClass
{
    public void MethodShared()
    {
        Console.WriteLine("This code is coming from shared
assembly");
    }
}
```

5. The shared assembly created needs to be stored into GAC where all the shared assembly in .NET are listed. Hence, rather than taking reference to the DLL to the console application, we need to list it to GAC first.

6. We will then create a folder in a local folder, in my case I have created a folder in c : and named it `keys`.

Open **Visual Studio Command Prompt** and type the following command to generate a cryptographic key pair:

```
sn -k "C:\keys\sharedassemblykey.key"
```

The console will say **Key pair written to c:\keys\sharedassemblykey.key**.

7. Now go to Visual Studio and open the `Properties\AssemblyInfo.cs` file of the project **SharedAssembly**.
8. Type in the following command:

```
[assembly: AssemblyKeyFile("c:\\keys\\sharedassemblykey.key")]
```

9. Replace the path of the file in the attribute. Also please make sure the filename is spelled correctly.
10. Now build the project. It will produce an assembly with signed keys. Open Visual Studio again and type the following command:

```
gacutil -I "C:\\.....\\SharedAssembly.dll"
```

When you run this command, it should say **Assembly successfully added to the cache**. If you see any permission-related issue, you need to re-open the command prompt in administrative mode. To do this, right-click on the command prompt and select **Run as Administrator**.

11. Now switching back to Visual Studio, let's add a reference to the shared assembly of the console project. Select **Add Reference** to the console application project. Browse to select the assembly file.
12. Rebuild the solution. You will see that the shared assembly is not created in the `bin\\Debug` folder, yet if you call the method from `MysharedClass`, it will be called correctly.
13. To add a satellite assembly, let's right-click on the console application and create a folder, naming it `Resources`.
14. Right-click on the folder and select **Add New Item** and select **Resources File**. Name the file `MyResources.resx`. Add some resources and click on **Save** in Visual Studio.
15. You can add the key (which is the identifier that needs to be used by the project to refer to the actual text written in **Value**).
16. Now go to command prompt again and type:

```
resgen "C:\\.....\\Resources\\MyResource.resx"
```
17. The command should say **Writing resource file.... Done**.

18. The `resgen` command will produce a binary resource file named `MyResource.resources`. Copy this file to the `bin\Debug` directory of the project. Now to create a satellite assembly from the resource file, open the command prompt again and move to the location where the `.resource` file and the actual executables are stored (in the `bin\Debug` directory). Type the following command:

```
al /t:lib /embed:MyResource.resources, MyAssemblies.MyResources.  
Resources /culture:en-US /out:MyResources.resources.dll
```

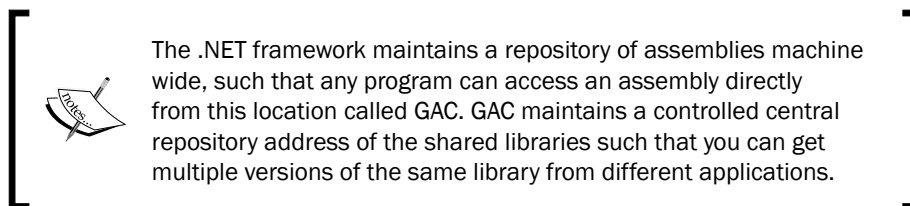
After the command is executed in the command line, it should not print anything on the screen other than the Microsoft welcome message.

19. After the DLL is created, create a folder and name it with a culture name. In our case I have created a folder named `en-US` and copied the resource file into it.

How it works...

Private assemblies are local assemblies that are entirely deployed for the usage of one single application. When the application loads and calls for an assembly, the memory for that assembly gets loaded. The application loads up the memory of the assembly during runtime into its own `AppDomain`.

Shared assembly is, on the other hand, useful to share an assembly between more than one application. There exists only one copy of an assembly in the GAC on one specific version and every application that loads up the memory for this assembly needs to load from the same location. The GAC is accessible to every application.



Satellite assemblies are assemblies that store localized resources. You can use a satellite assembly to store culture-specific codes that are based on the culture of the current application; the application can load up an appropriate assembly.

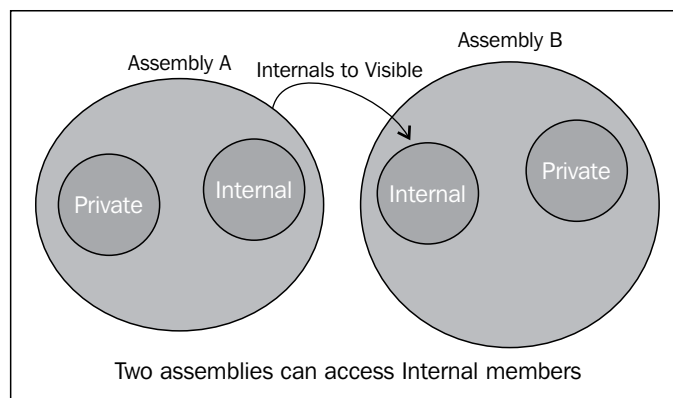
There's more...

There are a lot of small important facts that you should remember while working with assemblies in day-to-day life. Let's take a look at some of the other interesting topics related to assemblies.

Creating a friend assembly

It is important to understand that internal types are only accessible to the types that are present within the same assembly. That means internal types (known as friend types) are public within an assembly but not outside it. To make one type exposed to the external world, we need to make it public. But this does not solve some of the real-world scenarios. There might be situations that can arise which require you to access members from another assembly without leaving the type exposed to the external world. Say for instance, you want to separate the test code that runs on an assembly or you are developing an assembly that needs access to some component directly defined within another assembly. In these cases, friend assemblies are exceptions to the general rule. These assemblies expose any internal types defined within one assembly to another assembly which is defined explicitly.

An assembly can be defined as a friend assembly by using an attribute `InternalsVisibleToAttribute`, which specifies one assembly file to be friend to the assembly such that the internal objects defined within this assembly are accessible from the other:



In the preceding diagram, you can see that the private members remain inaccessible from outside the assembly even though the assembly is regarded as a friend but yet all other properties can be accessed directly.

You can specify the `InternalsVisibleTo` attribute either at the start of the source code or in the `AssemblyInfo.cs` file. Once the attribute is applied for a particular assembly, any class defined as internal, inside the assembly, will be accessible to a type on the other assembly.

Let's create two assemblies to demonstrate this problem:

1. Open Visual Studio and create a new a **Class Library** project and name it Assembly1.
2. Once the IDE loads up with a new blank class, we replace the existing code with the following one:

```
public class Assembly1Class
{
    private void PrivateMember()
    {
        Console.WriteLine("Private Member");
    }
    internal void InternalMember()
    {
        Console.WriteLine("Internal Member");
    }
    public void PublicMember()
    {
        Console.WriteLine("Public Member");
    }
}
```

The class creates three methods, each of which belongs to a separate access group.

3. Right-click on the **Solution** node in the **Solution Explorer** pane and add a new **Class Library** template, we will call it Assembly2.
4. Take reference of Assembly1 in Assembly2. Now let's write the following code:

```
Using Assembly1;
public class FriendClass
{
    public void TestMembers()
    {
        Assembly1Class oclass = new Assembly1Class();
        oclass.PrivateMember();
        oclass.InternalMember();
        oclass.PublicMember();
    }
}
```

The FriendClass class creates an object of Assembly1Class which is defined within the first assembly and tries to call its members.

5. If we try to compile the project, it puts two error messages saying that `PrivateMember` and `InternalMember` are inaccessible because of the protection level.
6. Now let's open `Assembly1Class` and put the following line at the start of the class:

```
[assembly: InternalsVisibleTo("Assembly2")]
```
7. If we compile the project now, only one error will still exist because of the private member. The `InternalsVisibleTo` attribute defines `Assembly2` to be a friend of `Assembly1`, so that it can access internal members in addition to the public members from outside.

If we want to use a signed assembly from GAC to be available as friend to an assembly, we need to specify the exact path of the assembly using the entire public key with the assembly name as shown in the following command:

```
[assembly: InternalsVisibleTo("Assembly2, PublicKey=99434...")]
```

We can specify the `InternalsVisibleTo` attribute inside the `AssemblyInfo.cs` file for a project as well.

How to delay sign an assembly

When we need to ship a DLL with an application, we need to sign an assembly so that it can be installed inside the GAC of the target machine and can be used up as a shared assembly when required. During the development process, where one or more groups of developers are working on the same assembly, there is a possibility that the private key of the assembly might be mishandled. But the development process needs to sign the assembly just to invoke the tests on it. We use **delay signing** on an assembly during the development phase by generating a partial signature during development to access only the public key. The private key remains secret and secure, and can be used only during the final build before shipping the project.

We will use the Strong Name tool to extract the public key portion of the actual key. Let us suppose the actual key for the organization is `keyfile.snk`, we then use the following command:

```
sn -p keyfile.snk myKey.snk
```

This command will extract the public key portion of the `keyfile.snk` file and put it into the `mykey.snk` file.

Once the key is created, open `AssemblyInfo.cs` under the project folder and put the following attributes:

```
[assembly: AssemblyKeyFileAttribute("myKey.snk")]  
[assembly: AssemblyDelaySignAttribute(true)]
```

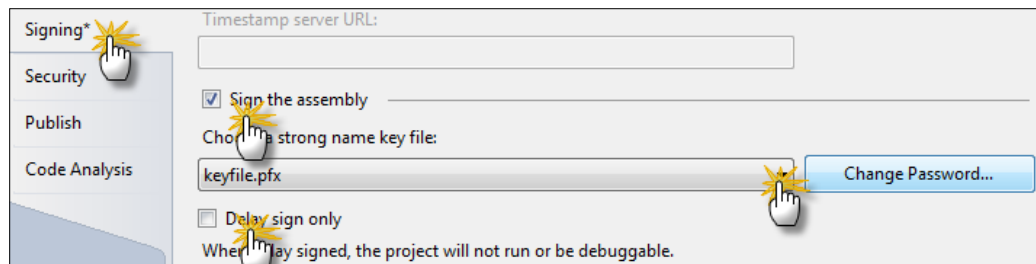
The first attribute passes the name of the file containing the public key, and the second attribute instructs that delay signing has been used for the current project. Once the file is put into the application directory, the compiler inserts the public key into the assembly manifest. Now as these assemblies do not contain the strong name signature, we need to turn off the verification for the assembly. We will use the following command to stop verification:

```
sn -vr myAssembly.dll
```

Finally when the assembly needs to be shipped, we replace the key with the actual key using the `-R` option in the Strong Name tool:

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

You can also specify the delay signing for an assembly directly from Visual Studio. Go to **Project | Properties** and click on the **Signing** tab as shown in the following screenshot:



As in the screenshot, you can create a file for the key, and if you have already created that file, you can specify the file for the assembly and select the **Delay sign only** checkbox.

How to share a private assembly between more than one application

From the previous discussion, it is evident that a private assembly needs the referenced assembly to have a local copy so that it finds the referenced assembly when the application is loaded. So, whenever you need to load the same assembly to more than one application, either both of the applications need to be copied to the same application directory when deployed so that it finds the assembly from both the applications, or it needs to be loaded from the GAC. If the application does not find the referenced assembly to this location, it will create an exception and shut down. But sometimes we need to put all the assemblies into a common folder, and each application that is deployed to its own folder needs to reference the assembly and hence share a common assembly file together. This needs a manual load of the assembly. You can use code to load the assembly whenever the application is launched.

During the initial load up of the assemblies, the application searches for the initial application directory or GAC to load the assemblies that it has reference to. If the application does not find the assembly in the desired location, the `Application` object raises an `AssemblyResolve` event. We can use this event to load the missing assemblies to the application.

1. Let's say we place the assembly that needs to be shared, with more than one application, to a separate directory named `C:\Assemblies`, and the name of the assembly is `MyPrivateSharedAssembly.dll`. Now to load up this assembly to an application we need to create an application.
2. Open Visual Studio and create a new application.
3. Subscribe the event handler for the `AssemblyResolve` event during the start up of the application. We use the following code to hook up the `AssemblyResolve` event:

```
AppDomain.CurrentDomain.AssemblyResolve += new ResolveEventHandler  
(CurrentDomain_AssemblyResolve);
```

4. Once the event is hooked up, we need to load the assembly inside the event handler using the following code:

```
static System.Reflection.Assembly CurrentDomain_  
AssemblyResolve(object sender, ResolveEventArgs args)  
{  
    string[] assembliesFailed = args.Name.Split(',');  
    string defaultPath = "c:\\Assemblies";  
    string assemblypath = Path.Combine(defaultPath,  
assembliesFailed[0] + ".dll");  
    if (File.Exists(assemblypath))  
        return Assembly.LoadFile(assemblypath, Assembly.  
GetExecutingAssembly().Evidence);  
    else  
        throw new ApplicationException("Assembly not found");  
}
```

5. When the application loads up, if any of the assembly is found missing inside the application directory (or bin directory), it will automatically invoke the event handler and get the assembly from the default path specified.

You can use this code in multiple applications to load the same assembly file for each of them.



When an application loads the assembly using `Assembly.LoadFile`, the assembly remains locked and cannot be modified from any other application. If you need the file to remain unlocked, you can read the file using a byte array and load the Byte array:

```
byte[] assemblyBytes = File.ReadAllBytes(assemblypath);  
return Assembly.Load(assemblyBytes);
```



Replacing `Assembly.LoadFile` with the preceding one will solve the problem of locking the file.

Also, it is important to note that even though the assembly resides in same directory, the memory allocated for the assembly will be different for each process that has been launched for the application. In other words, the assembly memory is created once per application (process).

How to use the Global Assembly Cache tool

As we already know that the GAC maintains a central repository of all the assemblies that are loaded into the machine with their specific public key token and associated version information. So whenever an application calls for a specific DLL, the GAC can easily point to the appropriate one and load the DLL to the application.

Even though we generally load a `.dll` file into the GAC using an installer, if we manually want to manage DLLs inside it, we need to use the Global Assembly Cache tool. The GAC tool (**Gacutil**) is actually a command-line utility tool that helps in checking, installing, or uninstalling a library from the GAC.

To use this tool, open **Visual Studio Command Prompt** and use the command-line utility.

One can check the availability of a shared assembly using the following command (in our example, the name of the assembly is `myassembly.dll`):

```
gacutil.exe /l myassembly.dll
```

This command will list all the assemblies that are found on the GAC with the count specified.

If you want to install an assembly in the GAC, we use the following commands:

```
gacutil.exe /i myassembly.dll
```

```
gacutil.exe /u myassembly.dll
```

The two commands will install and uninstall `myassembly` into the GAC respectively.

You can use the command argument `/?` to get the list of all command options available with Gacutil.



Even though GAC is not actually a folder, it is actually a tree of folders on which all the shared DLL is stored. Each DLL is stored in its own folder. The main location of the GAC in hard disk is located at `%windir%\assembly\`. The .NET 4.5 GAC is located at `%windir%\Microsoft.NET\assembly\`.

Inspecting the major components of a .NET program

When writing a program in .NET, it is not always true that the entire program that you create contains only code, but rather there are a lot of things in addition to that which exist besides source code. Some of the major components are the manifest, configuration files, information file, metadata, and so on.

Every program generally needs to communicate with the underlying operating system in some way. The manifest file contains all the metadata information needed to specify the assembly's version requirements and/or security identity and all information regarding the scope of the assembly. An assembly can contain the manifest file internally or externally. When the manifest file exists externally, we call it as a standalone manifest file, while the data can be incorporated inside the PE file as well.

The configuration and settings file on the other hand stores the application-specific or user-specific settings for the application. The configuration files are meant to be modified after the application is deployed, so they are stored as a `.config` file directly in the application directory with the same name as the executable. Sometimes when you need your configuration file to store sensitive information, we can also use encryption to encrypt the data inside it.

Finally, the assembly information file. This file is required by every application and the information that we put inside the file is compiled and stored as header information or metadata information of the assembly. When you create an assembly inside Visual Studio, a separate file is automatically created inside the project folder which holds information about the assembly such as product name, description, trademark copyright, and much more. This information will become a part of the assembly when it is compiled.

In this recipe we will try to look at the associated information regarding an assembly and how they are shipped to the actual assembly.

How to do it...

To understand the different sections of a program, let us follow these steps and create a simple application:

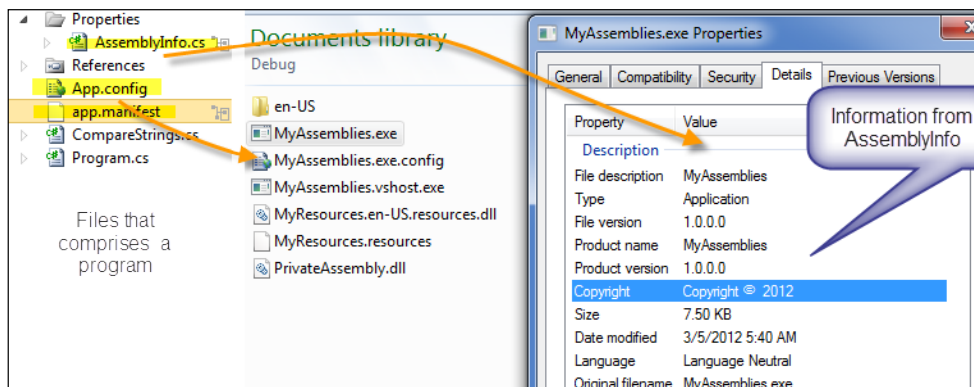
1. Start Visual Studio and create a new **Windows Application** project.
2. You will be shown a blank window designer when the project is loaded. Let's add a new configuration file (if it is not added already) with the name `app.config`. Right-click on the project and select **Add | New Item**. From the new item dialog box select **Application Configuration File** and click on **OK**.

Have a look at the following code:

```
<connectionStrings>
  <add name="MyConnectionString" connectionString="Data
Source=(local)\SQLExpress;Initial Catalog=DatabaseName;Persist
Security Info=True;User Id=sa;pwd=pktsa!" providerName="System.
Data.SqlClient" />
</connectionStrings>
  <appSettings>
    <add key="CurrentSetting1" value="data1" />
    <add key="CurrentSetting2" value="mystring2" />
  </appSettings>
```

In the preceding code we have created one `connectionStrings` key with name `MyConnectionString` and two settings keys. You can use the `ConfigurationManager` type to access these settings.

- Now right-click on the project again and add a new item. From the dialog box, we choose **Settings File**. The settings file is used to store user information and settings. Basically the settings file once saved actually stores everything in `config` file during the debugging session, (after it is deployed it will put it inside the user settings location). Let's add a few settings inside the UI. You will notice that the settings file actually creates a type to access this configuration, and hence settings are strongly-typed configurations.
- Add an **Application Manifest File** template in the same way as for the other two files. The application manifest file comes with existing code written already inside it. The most common usage of an application manifest file is to define the assembly identity version or the trust level that needs to be applied to the executable after it is deployed.
- The assembly information file should have already been added to the solution after the project was created. If you open the `Properties` folder inside the project, you will see the `AssemblyInfo.cs` file. Open the file, and you can see all the assembly-level attributes already mentioned with their default names:



In the preceding screenshot, the **Solution Explorer** pane is depicted with a number of files that form the major component of a program. You can see the `AssemblyInfo` file under the **Properties** folder, the information of which is deployed inside the assembly file, the configuration file (`App.config`) is deployed as `program.exe.config` and the `app.manifest` file which stores the required manifest information.

How it works...

In this recipe we have addressed the four types of files that comprise every assembly. Let's take a look at them one by one.

Entry point

The most important part of a program is its **entry point**. Every assembly defines the entry point which identifies from where the executable starts execution. The entry point of a program is generally defined in its main method. There are a few variations of the main method that are allowed in .NET:

```
static void Main() {...}
static int Main() {...}
static void Main(string[] a) {...}
static int Main(string[] args) {...}
```

Each of these methods can be defined only once and when the application gets compiled to an assembly, one of these methods will have an entry point defined which starts the application. You should note that the entry point is not mandatory for an application. For instance, the assembly that acts as a class library generally does not hold an entry point.

The AssemblyInfo file

The `AssemblyInfo` file is a repository of all assembly-related information which is compiled to the assembly and later exposed from the assembly file defining specific information about it. There are a number of defined attributes that you can assign to an assembly and a few of them are exposed externally as metadata of an assembly. For instance, if you open the `AssemblyInfo.cs` file, you will see a lot of interesting attributes that are already defined inside the assembly. Let us look into some of the most important attributes:

- ▶ `AssemblyTitle`: This defines the title of the application
- ▶ `AssemblyDescription`: Using this you can provide description about the assembly
- ▶ `AssemblyProduct`, `AssemblyCompany`, `AssemblyVersion`, `AssemblyFileVersion`, `AssemblyTrademark`, and so on: These are some of the other interesting configurations that you can specify

For specifying general metadata configuration of the assembly, there is another attribute that has been added to the .NET 4.5 base class library called **AssemblyMetadataAttribute**. It is a general purpose assembly attribute that allows associating a key-value pair to an assembly. There can be a number of these metadata components which can be associated with an assembly. Say for instance, you want to specify the home page URL of the product company that builds the assembly—the `AssemblyMetadata` attribute can be used to indicate this inside an assembly.

The application manifest file

The application manifest file is another important consideration when dealing with security and privileges that are needed by the application when run in a Windows environment. Every application in .NET certainly contains a certain level of privilege set when run inside a Windows environment. Operating systems higher than Windows XP do not allow an executable to run in the admin mode when being launched; if not, it asks for it when it is launched. The application manifest file allows the .NET application to request a certain level of permission when the application is launched. The application manifest file also describes the metadata for files that are private to the application. The application manifest file is embedded inside the .NET assembly as a resource. Let us see what the application manifest file looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
  <security>
    <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
      <requestedExecutionLevel level=" highestAvailable"
uiAccess="false" />
    </requestedPrivileges>
  </security>
<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">
  <application>
    <!--This Id value indicates the application supports Windows
Vista functionality -->
    <supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}"/>
    <!--This Id value indicates the application supports Windows 7
functionality-->
    <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>
  </application>
</compatibility>

  <assemblyIdentity type="win32"
    name="PKTPub.Experts.dotnetSample"
    version="6.0.0.0"
```

```

        processorArchitecture="x86"
        publicKeyToken="0000000000000000"
    />
    <dependency>
        <dependentAssembly>
            <assemblyIdentity type="win32"
                name="PKTPub.Research.SampleAssembly"
                version="6.0.0.0"
                processorArchitecture="X86"
                publicKeyToken="0000000000000000"
                language="*"
            />
        </dependentAssembly>
    </dependency>
</assembly>

```

Here in this XML manifest file, the first section defines the privileges that the executable should consider when the application is being executed. We have used `highestAvailable` to ensure that the application runs on the highest permission set available in the current environment. The other options that you can specify include `AsInvoker` and `requiresAdministrator`. Depending on the existing privileges, the application generally asks with a prompt whenever the administrator permission is required to execute. The `compatibility` node defines the operating system that can run the executable. The GUIDs specify the respective operating systems. Finally, the dependency of private assemblies associated with the current assembly is defined.

The application configuration file

The final and the most important component that every application needs is the application configuration file. An application configuration file is stored externally and holds special configurations about the application that can also be configured or changed later (even after the application is deployed). The application configuration (`app.config`) file is deployed as an `application.exe.config` file. The `app.config` file holds sections that in turn hold special information about assemblies and options used inside each assembly. The most common and usual usage of application configuration files is the `connectionStrings` and `application settings`. Let us look into it:

```

<connectionStrings>
    <add name="MyConnectionString" connectionString="Data
Source=(local)\SQLExpress;Initial Catalog=DatabaseName;Persist
Security Info=True;User Id=sa;pwd=pktsa!" providerName="System.Data.
SqlClient" />
</connectionStrings>
<appSettings>
    <add key="CurrentSetting1" value="data1" />
    <add key="CurrentSetting2" value="mystring2" />
</appSettings>

```

The `connectionStrings` and application settings are directly available inside the application using `ConfigurationManager`. So if we use `ConfigurationManager.AppSettings["currentSetting1"]`, it will give us `data1` as output. Similarly if we use `ConfigurationManager.ConnectionStrings["MyConnectionString"]`, it will give the whole `connectionString` value as output. `ConfigurationManager` also exposes APIs that help in getting sections on a particular configuration.

There's more...

There are a few operating system-specific configurations that you might also try to implement from your program.

How to work with custom configurations for an application

Configuration is one of primary concerns that you need to keep in mind while building an application. Irrespective of whether it is an ASP.NET web application or a Windows Application or a Windows Service, a configuration file is generally needed. We create configuration files in XML formats and the compiler automatically binds them with the executable as `exe.config`. This is later loaded by the program when it is being executed. A configuration file is a sequence of configuration sections which individually hold a format of the configuration. Some of the configuration sections are already built by the framework and are generally used. Some of the widely used configuration sections are `connectionStrings`, `appSettings`, `system.Web`, and so on. Today we are going to build a custom configuration section and show you how it can be used for your daily programming needs.

While dealing with configurations, there are three things that you need to address:

- ▶ `ConfigurationSection`
- ▶ `ConfigurationElement`
- ▶ `ConfigurationElementCollection`

For the purpose of defining simple configuration sections, the first two classes are generally enough, but when you need more complex configurations for your application, say which itself holds a sequence of configurations, you might need to use `ConfigurationElementCollection` and store the collection of individual `ConfigurationElement` objects in a program. Here in this recipe we will see how to create a custom configuration file taking all of these classes.

Getting ready

In this recipe, we are going to implement a special configuration which stores information on the servers that the application needs to use:

```
<Servers>
  <Element name="LocalServer"
    servername="192.168.179.60"
    userid="abhishek"
    password="password"
    isactive="false" />
  <Element name="RemoteServer"
    servername="68.240.22.19"
    userid="abhijit"
    password="passcode"
    isactive="true" />
</Servers>
```

This two-server configuration has a number of settings such as `servername`, `password`, `userid`, `isactive`, and so on. If we use a normal configuration, we cannot store all this information at the same time, and it would be difficult to use `appSettings` to store it. We can use custom configuration to deal with such scenarios.

How to do it...

As we have already mentioned, to deal with these types of complex configurations, we need to implement three types: `ConfigurationElement`, `ConfigurationElementCollection`, and `ConfigurationSection`. Now let us write the code to form a configuration:

1. Create a class called `Element` and inherit it from `ConfigurationElement`. Each `ConfigurationElement` object represents a unique configuration, let us suppose in our case it maps to `Element`.
2. Specify the properties of each individual configuration attribute and specify the `ConfigurationProperty` attribute to each of them that directly maps to the configuration file. You can also specify a few properties for each of these properties:

```
[ConfigurationProperty("isactive", DefaultValue = "true", IsKey =
false, IsRequired = false)]
public bool isactive
{
    get { return (bool)base["isactive"]; }
    set { base["isactive"] = value; }
}
```

For instance, here the `isactive` property maps to the `isactive` attribute in the configuration element. You can see the `ConfigurationProperty` attribute has been applied to the property to indicate the metadata. The name "`isactive`" specifies the attribute name, while `DefaultValue` and `IsRequired` are special attributes that you can specify for a property. You can also specify whether the property is a key or not. Remember, XML is case sensitive and you can only specify one key per element.

3. Implement the `ConfigurationElementCollection` class to specify the collection of the elements. While implementing you need to specify the type on which the collection will map. We specify the `ConfigurationCollection` attribute with `Typeof(Element)` to specify our `ConfigurationElement` value.
4. The `ConfigurationCollection` attribute defines an indexer which indexes all the elements that are found in the configuration file. Remember to specify the return type of the indexer as `Element`. The `GetElementKey` method is overridden to return the `Keyfield` value for `Element`. We can also mention that the configuration is read-only or not from here. We also need to return a New element from the `CreateNewElement` method.
5. Finally, we wrap `ConfigurationCollection` from inside `ConfigurationSection`. This class returns the collection of servers to the application. The XML element has been also specified using `ConfigurationProperty` for this class too:

```
public class ConnectionSection : ConfigurationSection
{
    [ConfigurationProperty("Servers")]
    public ServerAppearanceCollection ServerElement
    {
        get { return ((ServerAppearanceCollection)
(base["Servers"])); }
        set { base["Servers"] = value; }
    }
}
```

6. Once the configuration has been built, we specify the configuration directly inside the application configuration file with proper `SectionType` (in our case it is `ConnectionSection`), we use the `ServerElement` property to enumerate all the elements:

```
public IEnumerable<Element> ServerElements
{
    get
    {
        ServerAppearanceCollection collection = (ConnectionSection)
ConfigurationManager.GetSection("serverSection").ServerElement;
        foreach (Element selement in this.ServerApperances)
```



```

        {
            if (selement != null)
                yield return selement;
        }
    }
}

```

7. In the preceding code, you can see that `serverSection` is found using the `GetSection` API call of `ConfigurationManager` and we cast the output into `ConnectionSection` to get our object. If everything works well, this cast will be valid.

How it works...

.NET configuration uses the same technique which is used in .NET serialization. **Serialization** is a technique of representation of a runtime object in some data format, such that the serialized data can again be deserialized to produce the actual object again. The whole configuration file has been separated into a number of sections, each of which are serialized XML representations to a valid .NET object. In this recipe, we have created a collection of custom configurations and mentioned the same using a section name. The section name is important to specify each section of the configuration:

The image shows an XML configuration snippet on the left and a screenshot of a Windows application window titled 'MainWindow' on the right. A blue arrow points from the word 'Type' to the 'type' attribute in the XML code.

XML Configuration:

```

<configSections>
  <section name="serverSection"
    type="ConfigurationSettings.ConnectionSection, ConfigurationSettings"/>
</configSections>
<serverSection>
  <Servers>
    <Element name="LocalServer"
      servername="192.168.179.60"
      userid="abhishek"
      password="password"
      isactive="false" />
    <Element name="RemoteServer"
      servername="68.240.22.19"
      userid="abhijit"
      password="passcode"
      isactive="true" />
  </Servers>
</serverSection>

```

MainWindow Table:

Name	Server Name	Is Active	User Id	Password
LocalServer	192.168.179.60	False	abhishek	password
RemoteServer	68.240.22.19	True	abhijit	passcode

Within our `configSections` tags, we specified the name of the configuration and the type which handles the configuration. Once we call `GetSection` from configuration, it reads the `config` file with the appropriate section and tries to cast it into the type we specified.

If configuration is valid, the cast is valid, and we get the exact representation of the section in a managed object representation.

There's more...

Creating a custom configuration and using it with an application looks very sophisticated. Even library developers put in a lot of configuration blocks that enable some hidden configuration of the library. The use of configuration has been growing day by day. Now, let us take a look on some of the additional options that you might sometimes require.

How to change the configuration of an application at runtime

We often come across a requirement to change the value of the configuration during runtime of the application. The `ConfigurationManager` API exposes methods that enable the application to write data at runtime.

First of all, you should note, that when the application is deployed, the `app.config` file would be modified to the actual `applicationName.exe.config` file. You can also open the config file using the XML document and replace it manually, but it is recommended to use `ConfigurationManager` to do this. Let's see how to do this using code:

```
System.Configuration.Configuration config =
    ConfigurationManager.OpenExeConfiguration(
        ConfigurationUserLevel.None);
config.AppSettings.Settings.Add("mykey", "myvalue");

// Save the configuration file.
customSection.SectionInformation.ForceSave = true;
config.Save(ConfigurationSaveMode.Full);
```

Here, we open the configuration during runtime using the `OpenExeConfiguration` method of `ConfigurationManager`. This is important because the configuration is actually deployed as a `exe.config` file. After the configuration is open, you can modify `appSettings`, `connectionStrings`, or even add `CustomSection` to the configuration and finally use the `Save` method to save it.

`ForceSave` ensures that the configuration is saved in the `config` file even if it is not modified.

How to deal with configuration versions

Configuration is one of the most important parts of a .NET program. Visual Studio 2012 comes with a feature that automatically detects appropriate configuration files to ship with the executable when the project is built with different modes. For instance, say when the project is built in the debug mode, we need a separate set of configurations. In the testing phase we need a staging configuration, and finally in the production release we need to ship another set of settings. Hence we really need to have more than one configuration stay for an application that needs to be deployed based on the state of deployment.

1. Let us create a new **Console Application** project and add two application configuration files, namely, `debug.config` and `release.config` to the project.

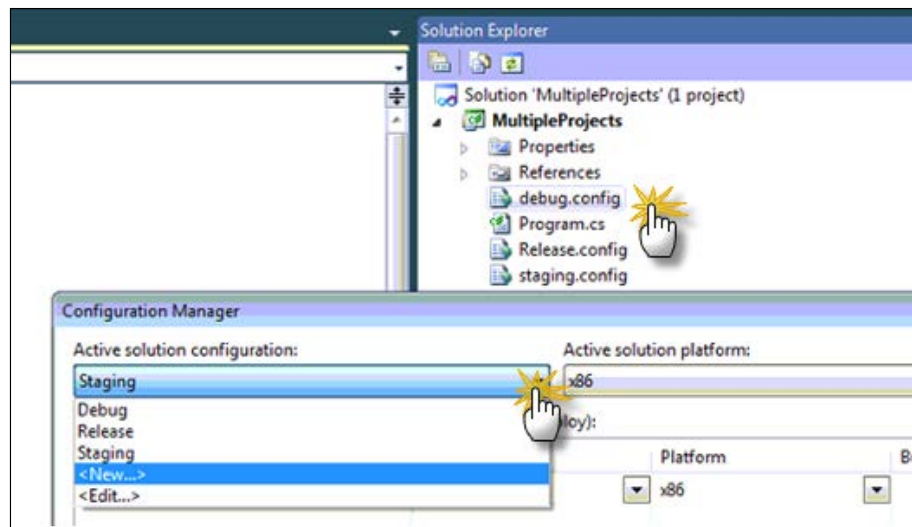
2. Put one configuration to each of them. I have used `<add key="myKey" value="This is from debug"/>` in `appSettings` of my debug and the same key with different messages in the release version of the application. In my release version the string is `This is from release`.
3. Now to use these configurations, right-click on the project in the **Solution Explorer** pane and select **Unload Project**.
4. Select **Edit project.csproj** and navigate to the node **Target**. If there is no target defined for the project, you will see that the configuration section has been commented out. Add the following code to the section:

```
<Target Name="AfterBuild">
    <Delete Files="$(TargetDir)$(TargetFileName).config" />
    <Copy SourceFiles="$(ProjectDir)$(Configuration).config" DestinationFiles="$(TargetDir)$(TargetFileName).config" />
</Target>
```
5. The preceding configuration will put a task at the `AfterBuild` event of MSBuild and delete the existing `config` files (if any) and copy the file from the project directory based on the name of the configuration file. Remember, as we have used `$(projectDir)$(Configuration).config`, it is important to name the `config` files in the project directory with the same name as the name of the build configuration (in our case we named it as `debug.config` and `release.config`).
6. Save the content and reload the project.
7. Now put a small code in the `Program.cs` file to invoke the configuration as follows:

```
static void Main(string[] args)
{
    string key = ConfigurationManager.AppSettings["myKey"];
    Console.WriteLine(key);

    Console.ReadKey(true);
}
```
8. When you run the code in debug mode, you will now see the configuration from debug and when in release mode, it will show the release string.

Also, in certain cases it is required to create more configuration builds than debug and release. In such cases you can use `ConfigurationManager` from Visual Studio IDE to create as many configuration stages as you want:



Here we have created a new configuration as **Staging** and added `staging.config` to the application.

How to disassemble an assembly

As you might already know, .NET assemblies are compiled to produce an intermediate language rather than a machine-dependent language. Anyone who knows the specification very well can easily reverse engineer the process of compilation and decompile the IL into specific languages thereby exposing the logic that has been written to create the executable. The reverse engineering technique has been present for a long time. The introduction to general purpose-free disassemblers made it a practice for architects to look inside an assembly inspecting things before actually using them. Some disassemblers can even show the equivalent source code in .NET languages. Therefore, it is very easy for a normal-level programmer to look into the exact logic and understand the various sections of code. Thus, many of the information about licensing of third-party software can be very easily compromised if the entire logic is exposed to the assembly. Visual Studio is shipped with a tool that can show the intermediate language of an assembly. By looking at the intermediate language, one can quite easily understand what logic is written inside an assembly and can reverse engineer it to its original source code. The tool is called **IL Disassembler**. In this recipe we will use this tool to demonstrate the IL equivalent of an assembly and try to understand the reverse engineering techniques.

Getting ready

Let us create a blank code to demonstrate this recipe. Open Visual Studio and create a **Console Application** project. Call it `DisassembleCode` and select **OK**. Let's write a few lines of code inside the `Program.cs` files and create a class to show the recipe:

```
class Program
{
    static void Main(string[] args)
    {
        string firstString = "Hi this is my first string";
        string secondString = "Hi this is my seconds string";

        CompareStrings cstrings = new CompareStrings();
        cstrings.FirstString = firstString;
        cstrings.SecondString = secondString;

        if (cstrings.Compare())
            Console.WriteLine("these strings are equal");
        else
            Console.WriteLine("These strings are unequal");

        Console.ReadKey(true);
    }
}

// CompareStrings
class CompareStrings
{
    public string FirstString { get; set; }

    public string SecondString { get; set; }

    internal bool Compare()
    {
        return this.FirstString.Equals(this.SecondString);
    }
}
```

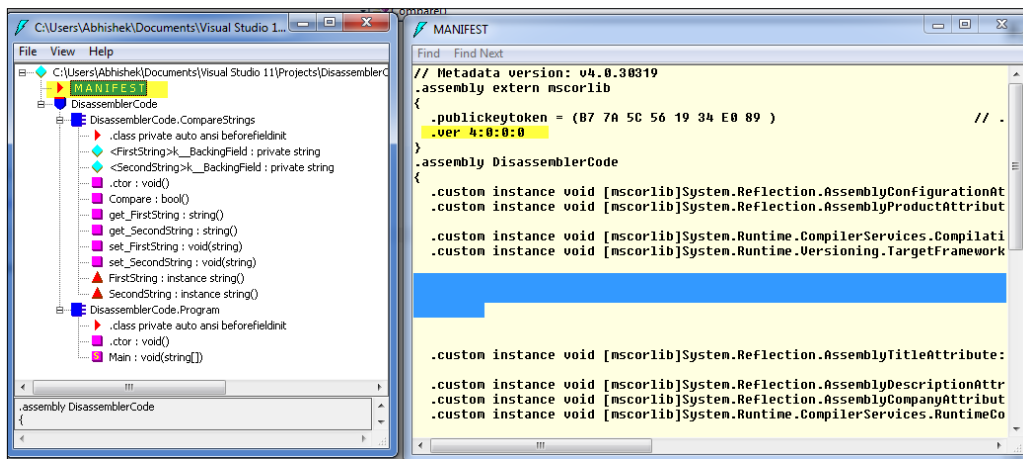
The preceding code just compares the two strings which are statically defined within the assembly and produces an output based on whether the strings are equal or unequal. Here in our code, it always results to unequal because the value of `FirstString` is not equal to `SecondString`.

Please note that the code introduces a new type called `CompareStrings` that initiates a new object within the main method and then invokes the `Compare` method to get the comparison result. When you compile the code, and go to `bin\debug` folder of the project, you will see an executable. When you run the executable, the application will open a blank console window and show the message.

Disassembling is a technique used to reverse engineer an intermediate code written inside an assembly to a human-readable code to understand the logic inside an assembly. Say for instance, you do not have the source code for the assembly and you want to know what exactly is written inside it. The recipe will focus on the use of the IL Disassembler that comes free with Visual Studio to see the various features of the program and how to know the entire structure of the assembly.

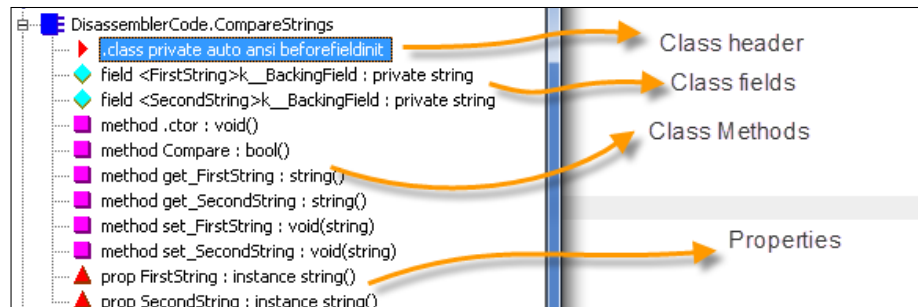
How to do it...

1. Open the folder containing the assembly. Generally the assembly is produced inside the `bin` directory of the application folder and produced with the same name as the project.
2. Open the IL Disassembler by going to **Start | All Programs | Visual Studio | Microsoft Windows SDK tools**.
3. Select **File | Open** or use the shortcut `Ctrl + O` from the IL Disassembler and select the executable that has just been created. If you don't remember the location of the assembly, you can copy the whole address of the assembly from the opened explorer and paste it in the **File Name** field of the **Open** dialog box.



4. When the assembly has been successfully loaded inside the IL Disassembler tool, it will produce a tree showing all the information about the assembly. You can see the tree shows the types, all the methods that are defined inside the assembly as a tree node, and also the metadata associated with the assembly.

5. When you double-click on any of the nodes, it will show the respective branch of code defined within the assembly. The preceding screenshot shows the metadata of an assembly, which generally translates the metadata compiled from the AssemblyInfo file and also gives insight about the version needed to run the assembly: the CorFlags, the public token, and so on.
6. Open the **DisassemblerCode.CompareStrings** node and check the various options available as shown in the following screenshot:



7. You can double-click on any node to see the corresponding IL code that is written inside the assembly.

How it works...

As you already saw various elements of the code inside the IL Disassembler, it is important to know a few basics of the IL construct to actually understand the basic structure of disassembled code. Let us check some of the interesting sections of an IL to understand the basic code structure:

```
.class private auto ansi beforefieldinit DisassemblerCode.
CompareStrings
    extends [mscorlib]System.Object
{

    .field private string '<FirstString>k__BackingField'
    .custom instance void [mscorlib]System.Runtime.CompilerServices.
CompilerGeneratedAttribute::.ctor() = ( 01 00 00 00 )

    .property instance string FirstString()
    {
        .get instance string DisassemblerCode.CompareStrings::get_
FirstString()
        .set instance void DisassemblerCode.CompareStrings::set_
FirstString(string)
```

```

} // end of property CompareStrings::FirstString

.field private string '<SecondString>k__BackingField'
.custom instance void [mscorlib]System.Runtime.CompilerServices.
CompilerGeneratedAttribute::.ctor() = ( 01 00 00 00 )

.property instance string SecondString()
{
    .get instance string DisassemblerCode.CompareStrings::get_
SecondString()
    .set instance void DisassemblerCode.CompareStrings::set_
SecondString(string)
} // end of property CompareStrings::SecondString

} // end of class DisassemblerCode.CompareStrings

```

Here in the preceding code, the first thing that you notice is actually the header information of the class. The `.class` stands for the start of a class (or type) in IL. Any type (other than a few exceptions such as Interfaces, and others) in a .NET environment is derived from `System.Object`. We do not specify this in our code, but you can see that the compiler automatically writes the information inside the IL. Notice that the `beforefieldinit` flag specifies that the type is "lazy initialized", or in other words, the type will be initialized only when it is really needed.

The next section creates an instance of a string. You can see each property has a compiler generated backing up the field as we implemented auto properties. `get` and `set` are respective methods that invokes the getter and setter of the property. The naming convention is `get_FirstString` and `set_FirstString` where IL is concerned.

Now let us demonstrate what the Compare method looks like:

```

.method assembly hidebysig instance bool
    Compare() cil managed
{
    // Code size          23 (0x17)
    .maxstack 2
    .locals init ([0] bool CS$1$0000)
//000015:      {
    IL_0000:  nop
//000016:      return this.FirstString.Equals(this.
SecondString);
    IL_0001:  ldarg.0
    IL_0002:  call     instance string DisassemblerCode.
CompareStrings::get_FirstString()
    IL_0007:  ldarg.0

```



```

IL_0008:  call      instance string DisassemblerCode.
CompareStrings::get_SecondString()
IL_000d:  callvirt  instance bool [mscorlib]System.
String::Equals(string)
IL_0012:  stloc.0
IL_0013:  br.s     IL_0015
//000017:  }
IL_0015:  ldloc.0
IL_0016:  ret
} // end of method CompareStrings::Compare

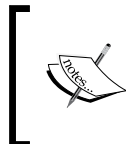
```

Here you can see that there are lot of things happening inside the IL for our `Compare` method. Each method in the IL comprises of three sections. The first one being the declaration where the local objects are referenced in a thread stack; second is the actual IL instruction code; and third is the return value.

Here in the preceding code, the highlighted section on the top defines the stack for the method. The main declaration initializes the size of the stack that can be used for the method. You can notice, even though we didn't define any local variable, the stack size still says 2 and defines a local Boolean variable to hold the compared values.

The IL instruction loads `Firststring` and `secondString` by invoking the `call` instruction to the respective getter method that has been already defined inside the code and finally uses `callvirt` to call the `Equals` method of the `System.String` type.

The instruction code `stloc.0` and `ldloc.0` stores and retrieves the local stack element respectively and returns back the already loaded value back from the method.



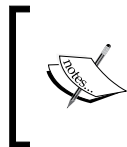
It is important to note that during the compilation process, the C# compiler does a lot of optimizations on code which sometimes gives different IL than that compiled from the source. In such cases the decompilation may result in a dissimilar source.

Let us look into some of the most important IL instructions:

- ▶ `.entrypoint`: If present, it will be present only once and defines the starting point of the assembly. You can find it inside the `Main` method.
- ▶ `.locals init`: This defines and initializes the stack locals for a method body.
- ▶ `stloc.*`: This stores the loaded value into stack location. `*` indicates the index of the variable that it is stored to.
- ▶ `ldloc.*`: This loads the location into memory. `*` indicates the stack location.
- ▶ `newobj`: This creates a new object instance by calling the constructor of a type.
- ▶ `call/callvirt`: This calls a virtual method on a specified object or type.

There are a lot of other IL instructions that you can look at and understand. You can get a list of all IL instructions from the following reference:

http://en.wikipedia.org/wiki/List_of_CIL_instructions



You can use `[assembly:SuppressIldasmAttribute()]` to suppress an assembly to be opened using the **IL DASM** tool. When you try opening an assembly with this attribute, it will show an error message: **Protected module**.

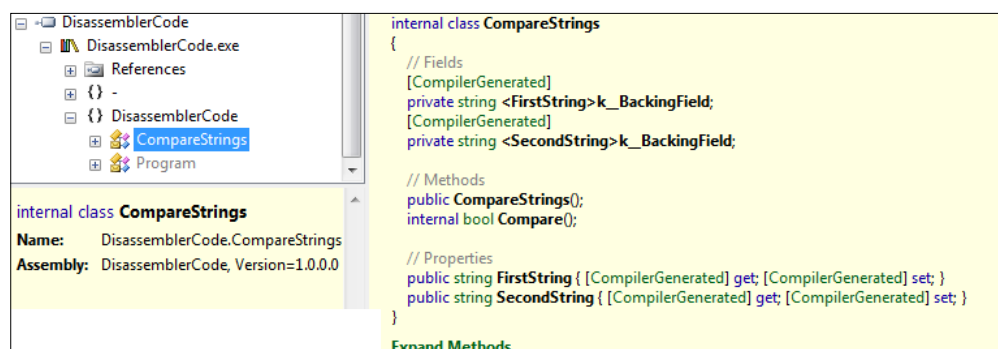
There's more...

It is not really a good idea to use only IL Disassembler to disassemble an assembly. There are some really interesting tools that can help in addition to the use of IL Disassembler. Let us look into them.

How to disassemble an assembly using Reflector

There are other tools that are smarter than the IL Disassembler which can readily convert the assembly code into their corresponding source code in any .NET language. The tool is really interesting and just one or two mouse clicks can convert the entire assembly into its corresponding source code equivalents. The .NET Reflector along with a few other tools are already available on the market for free which act as a first-class citizen to reverse engineer code. Let us see how the Reflector can help in converting existing code. You can download Reflector from <http://bit.ly/NETReflector>. Reflector 6.8 is free but higher versions are chargeable.

Open the Reflector tool (after you have downloaded and installed it in your system). Select **File | Open Assembly** and select an assembly. You can also use **Open Global Assembly Cache** command to list the assemblies installed in GAC. After the assembly is loaded into the Reflector tool, you will see the name of the assembly in the left-hand side as a node in the tree. Expand the tree to see the classes defined within the assembly:



In the preceding screenshot, you can see that the assembly is already loaded in the left-hand side of the screen, and the right-hand side shows the corresponding code for the selected class file.

The tool is basically reading the IL of the assembly and writes the corresponding high-level source code equivalent for it. It is evident that the conversion of the IL from the source code is done by the compiler. The tool reverses the technique and places the logic to reverse engineer the IL code to a source code.

So, from the drop-down list, if you select **C#**, the source is converted to C# and if you select VB .NET, you will see it is converted to VB. NET. The navigation between different class files is also very easy in this tool. You need to just click on the method or type and the corresponding code block will be displayed immediately.

How to merge assemblies into one

Each .NET assembly contains individual modules of code that work independently. An application needs to load each of those modules into the memory to call them when needed. So generally, it is always a good idea to separate source codes into assemblies to produce independent modules. Or in other words, separating two modules into separate assemblies means they should be independent of each other.

Sometimes it comes as a requirement to separate dependent modules into two or more assemblies. Say for instance, the size of the dependent assembly becomes so big that it is hard to maintain, or some separate groups of people are working on a product and they build code in different .NET languages. In such cases, it is important to separate the assembly into multiple disk files even though they are interdependent from one another. But loading two dependent assemblies into the memory puts additional pressure on the application and hence decreases performance. **ILMerge** is an interesting tool that helps to merge two or more assemblies into one, so that dependent assemblies are not separated into more than one disk file.

To merge two assemblies into one, we use the ILMerge tool which can be downloaded from <http://research.microsoft.com/en-us/people/mbarnett/ilmerge.aspx>. Once you download the tool, install it and you will find the executable somewhere inside C:\ProgramFiles\Microsoft\ILMerge\. Open the command prompt and move to the directory and try the following command:

```
ilmerge /target:winexe /out:CompositeProgram.exe    MyApplication.exe  
ClassLibrary1.dll ClassLibrary2.dll
```

The preceding command will merge three assemblies, MyApplication.exe, ClassLibrary1.dll, and ClassLibrary2.dll into one composite assembly CompositeProgram.exe. The target specifies what type of executable you are building in the target assembly. In our case we have used WinExe to ensure that it creates a Windows executable. If you specify target:exe in the command line, it will produce a executable or target:library for a DLL file.

What is IL Weaving

IL Weaving is a technique to inject IL code inside an assembly. As we already know, .NET when compiled produces a byte code, which is machine independent and need to be recompiled again using the JIT compiler to produce runtime machine-dependent code. Thus, the assembly that has been created using initial compilation can be easily changed and can be made perfect for JIT compilation. IL Weaving is the concept of either changing the IL code after compilation or changing it during the compilation process. For instance, Simon Cropp has produced one IL Weaving tool that automatically plugs in as an extension to Visual Studio and works as a post-build script to wire up the `PropertyChanged` notification to each individual property that needs it. The `Notify Property Weaver` runs as an MSBuild task.

You can try this with its entire source code from the following link:

<http://bit.ly/ILWeaver>

Securing your code from reverse engineering by using obfuscation

It is sometimes very important for an application to hide the code that is written inside a managed assembly. Using a managed environment, as we have seen in our previous example, it is very easy to decompile a code and hence reverse engineer the code and produce the exact same code that has produced the output assembly. We cannot hide the source code from an assembly because the intermediate byte code has equivalent source code equivalents. So for an expert it is really easy to understand the logic that is written inside an assembly. With the large number of .NET decompilers available in the market for free, it is very easy even for a basic developer to reverse engineer the source code. So when you really need to protect your source, for instance, by writing an algorithm to detect licensing of a product or even an algorithm that are sensitive or under **Non Disclosable Agreement (NDA)**, you need to only rely on obfuscation tools in a managed environment.

Obfuscation is a technique of hiding the meaningful code into something that is really confusing and hard to interpret. Hence, even though the source IL does exist in the assembly, they are written in such a way that neither a basic developer nor an expert could understand it. The use of control flow obfuscation, data obfuscation, renaming types, and so on, can really confuse and even make it impossible to understand the actual code.

Getting ready

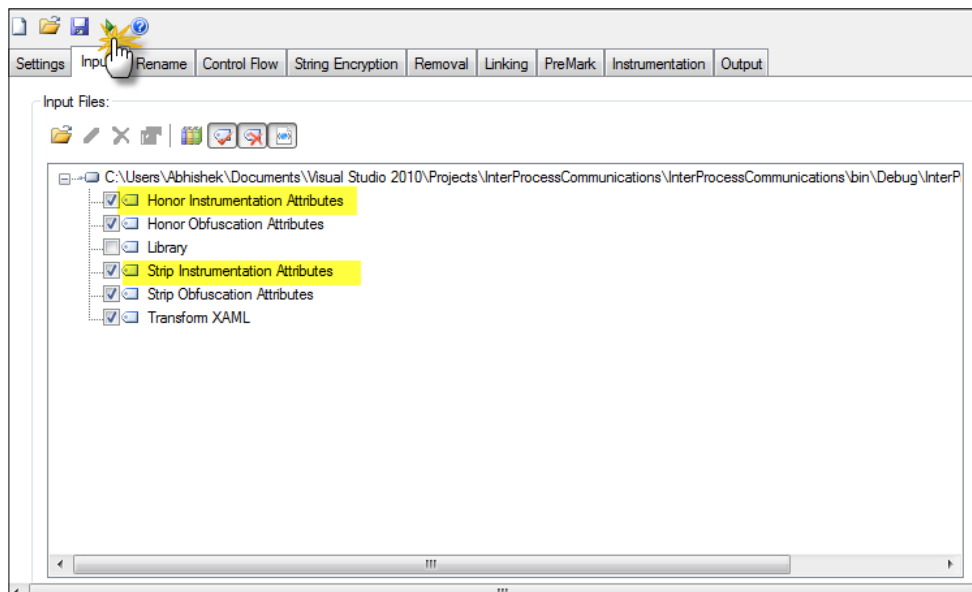
In this recipe, we are going to use the same code as we defined in our previous recipe but here I will be using **Dotfuscator** (<http://bit.ly/DotFuscator>) to obfuscate the code before looking into the code using IL Disassembler.

So to go with the recipe, let's create a console application and write in some code. Compile the project and create an assembly. Once the assembly is created, let us open the Dotfuscator Community Edition or Professional Edition and try obfuscating an assembly.

How to do it...

Let us follow the steps to obfuscate an assembly:

1. Open Dotfuscator and create new project. A project is an XML file that creates the necessary settings so that when you open the same project subsequently, everything will be loaded automatically as saved before.
2. Once you open the tool, the first thing that you notice is **Input Assemblies**. A special list is showed up on the tool that contains all the assemblies you choose to disassemble.
3. Input an assembly using the **Input Assemblies** node of the tree. Choose the appropriate assembly that you want to use. Remember, you can choose all the dependent assemblies of a project in one shot so that you can reduce the amount of time to create related projects and do necessary settings for every assemblies.



4. After the assembly is chosen, you can select the node on the list and select options. There are few options available such as **Honor Instrumentation Attributes** which indicates the tool to process these special attributes supported by Dotfuscator and perform the indicated transformation on the target assembly. Similarly, **Strip Instrumentation Attributes** strips all the Dotfuscator attributes from the final output.

5. The **Library** mode when selected will produce the output as a library that needs to be referenced by other assemblies. If you choose this, the publicly exposed objects do not rename it.
6. Once you are done with this, you can directly use the **Build Project** button to create the obfuscated assembly. If the project has not yet been saved, the tool will ask you to save the project as an XML file. This file is used to open the configuration again in future.
7. Once the build is complete, the obfuscated assembly will reside on the working directory from where you choose the assembly from inside a new folder called DotFuscated.
8. Once you save the project file, you can invoke the Dotfuscator directly from Visual Studio as well as using the command-line utility. Right-click on the project and select **Properties**. Go to **Build Events** and paste the following code inside **Post-build Event Command line**:

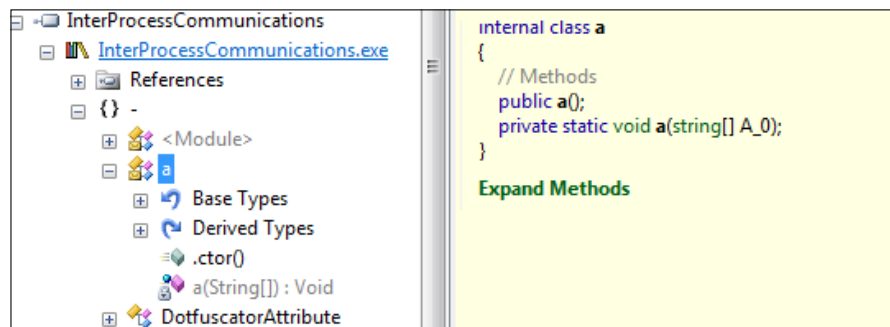
```
"C:\Program Files\PreEmptive Solutions\Dotfuscator community
Edition 2.0\dotfuscator.exe" "c:\DotfuscatorProjects\
dotconfiguration.xml"
```

9. Here dotconfiguration.xml is the project file from the Dotfuscator tool. You can also specify the configuration directly from the command line without using configuration file. For instance:

```
dotfuscator.exe /in: $(OutDir)\ $(TargetFileName) /appliance /
debug:on /break:on /enhance:on /clobber
```

This command when placed in the **Post-build Event Command Line** window, will compile the assembly as a .exe file, break ILDasm, produce PDB files, use enhanced overload induction, and silently overwrite the map file (clobber).

The assembly when produced will have every API renamed, hence when you try to reverse engineer the assembly using Reflector, you will see something like the following screenshot:



You can see that every property and method has been renamed with junk characters.

How it works...

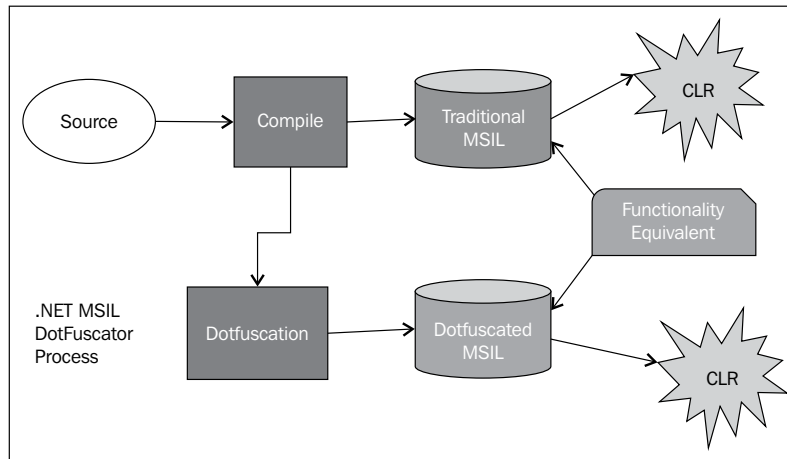
Dotfuscator is one of the most popular tools for obfuscating an assembly. Most people use this tool as a community version that comes for free with the Visual Studio installation. If you open the project file, that we just saved while performing obfuscation, we see something like this:

```
<input>
  <loadpaths />
  <asmlist>
    <inputassembly refid="0b86956e-8677-4887-8c18-7f24b0077ae8">
      <option>honoras</option>
      <option>stripoa</option>
      <option>transformxaml</option>
      <file dir="C:\Users\Abhishek\Documents\Visual Studio
2010\Projects\InterProcessCommunications\InterProcessCommunications\
bin\Debug" name="InterProcessCommunications.exe" />
    </inputassembly>
  </asmlist>
</input>
<output>
  <file dir="{configdir}\Dotfuscated" />
</output>
```

Even though the actual project file is longer than what you see, there are a few key things that you need to notice here in this file. The section that I have just opened here contains a section of `input` and `output` configurations. Clearly the `output` section only includes the file directory path which is specified as `{configdir}`, which is the actual assembly path with the `Dotfuscated` folder. You can change the directory path of the output to anything relative to `configdir` here.

The `input` section in addition to the file directory path, includes few additional options. `honoras` specifies the honor obfuscation attributes or `stripoa` specifies the strip obfuscation attributes.

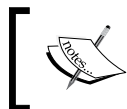
The entire process of obfuscation works as shown in the following diagram:



This diagram shows how the process of obfuscation takes place using Dotfuscator. The source code is compiled to produce Traditional MSIL, but we use Dotfuscation to obfuscate the MSIL to dotfuscated MSIL. If you see the assembly after obfuscation, you will find that a special namespace has been added for Dotfuscation. But the important thing to note here is that the functionality of both the obfuscated code and the normal code remains the same even though both represent CLR.

There's more...

Even though the Dotfuscation looks like it's working great, as all the types, members, or fields have been renamed, it is not exactly very secure. The code inside the obfuscated modules is still visible using popular disassembler code, yet it has been slightly difficult to understand. Any smart developer can still reverse engineer the code to get the actual logic. But code obfuscation goes well beyond simple renaming of types, members, or variables. There are features such as **Use Enhanced Overload induction** (only in non-commercial versions) which will rename many methods to the same name and hence will make it more difficult to obfuscate. The commercial version of Dotfuscator allows you to add a lot of other options that really help to secure the assembly code in such a way that it would be impossible to reverse engineer the code by hand. Let us look at some of these options.



You can download the trial version of a commercial edition directly from the Preemptive site (www.preemptive.com).

Using different additional options on Dotfuscator

Dotfuscator includes lot of additional options that you might use when you are using the Professional edition. Let's quickly demonstrate these options so that it is easier to jump into it if you really need to:

- ▶ **Renaming:** Renaming is an option that has already been discussed. It includes renaming of types, methods, and fields. Dotfuscator can also be configured to use **Overload Induction** to rename types with the same name when it can.
- ▶ **String encryption:** Strings are generally placed inside the code insecurely and even inside the IL. String encryption of obfuscation lets the strings being encrypted be stored in a special location where the program asks for it by its serial number.
- ▶ **Control flow obfuscation:** This option changes the control flow of the application logic and makes it non-deterministic and semantic making sure the output remains the same. In its options you can specify the control flow level as well.
- ▶ **Debugging Obfuscated code:** Generally, obfuscation changes the entire assembly so that every line in the original assembly will either be shifted to another location or the entire type will be renamed. Thus, it is impossible to use the PDB file that Visual Studio generates to debug the obfuscated DLL. Dotfuscator allows you to specify options to include PDB files during the obfuscation to allow debugging of the assembly. On the properties page, you can select **Emit Debugging Symbols** to generate debugging files.
- ▶ **Pruning:** This option statically analyzes the entire assembly and determines which portion of the code is not been used. Intelligently, it removes all the codes that are not used and produces the assembly.
- ▶ **Instrumentation:** This new feature of Dotfuscator allows you to generate reports on features usage of the assembly. Add custom attributes to the Dotfuscator project and the code will get automatically injected into the assembly to produce feature usage of the assembly and gives analytical data. In the **Instrumentation** tab, right-click on the executable selected and use the **Add attributes** to add these features. For instance, `BusinessAttribute` determines the information about the company that builds the assembly. You can specify the company name and company key. `ExceptionTrackAttribute` tracks the exception that occurs on the assembly. This is very important to build analytical data on what is actually producing bugs on the piece of software. You can also specify attributes to methods, types, and so on, similar to this.

Configuring an assembly directly from Visual Studio

Dotfuscator integrates with Visual Studio. When you install the Dotfuscator professional edition, you will get a template to create Dotfuscator projects. Let's create a **Console Application** project and add a new project to the solution, and select **MyDotFusculatorProject**. You will see that the entire tool has been loaded inside Visual Studio, and you can use the **Solution Explorer** pane to invoke specific UI elements.

Let's follow the steps to obfuscate directly from inside Visual Studio:

1. Once the project has been loaded, you can either choose the individual assemblies from the **Input files** folder or you can right-click on the **Solution Explorer** pane and select **Add Project Output**. This will prompt you to the **Add Project Output** dialog box. Choose **Primary Output** for each of the assemblies. You will see that each of the DLLs will get added to the **Input** folder inside the solution.
2. Choose the necessary configuration you want, in the same way you did for the tool.
3. Right-click on the Dotfuscator project and select **Properties**. Interestingly, this opens the **Properties** window to enable/disable each option that you want to use inside Dotfuscation. Generally, it is good to enable **Renaming**, **Control Flow**, and **String Encryption** to make a solid obfuscation.
4. Select **Build Settings** to ensure the **Output** directory has been configured correctly.
5. Build the solution, and you will get the output inside the **Output** directory.



You can select a lot of other options from the **Property Pages** window. For instance, if you change **Break ILDASM** to **Yes**, the Ildasm tool cannot open the assembly.

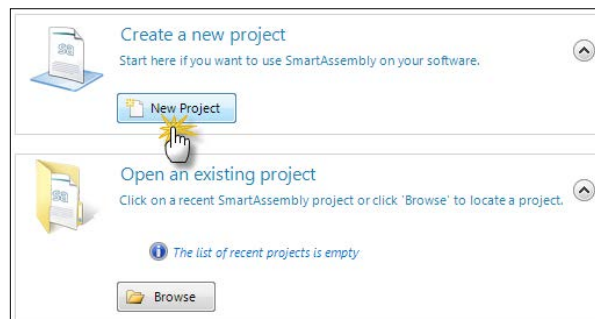
Similarly, if you change **Build Progress** to **Verbose**, it will give additional information on the **Output** directory of Visual Studio.

You can also change the **Emit Debug Symbols** property to **Yes** to output PDB files with the output assembly. Debugger can use this file during debugging sessions.

Obfuscation using SmartAssembly

SmartAssembly (www.smartassembly.com/) is a great obfuscation tool from Red Gate that works best if you want to purchase a commercial obfuscation tool. Let us take a look at the advantages of SmartAssembly and its ease of use.

Open SmartAssembly, in my case I am using SmartAssembly 6.0. In the initial screen, you can choose either to create a new project or open an existing project. In our case, as we want to do a new project we select **New Project**:



A new project opens up another screen which indicates the **Browse** option to select the assembly you want to obfuscate. We are going to select one of our existing assemblies to try the obfuscation using this tool. We also need to select the destination filename after the obfuscation is done. Once you do the initial setup, SmartAssembly opens the actual workspace with options:



Obfuscation with SmartAssembly is much easier than that with Dotfuscator. It depends on your choice and which one you use.

Understanding .NET garbage collection and memory management

.NET memory management is actually not something we generally need to bother about. Any .NET application that runs, actually contains its own garbage collector which manages the memory used by the program and releases the memory when it is not required. There is a high-priority thread that runs under every process which is called the **finalizer thread**. This invokes itself automatically when there is high memory pressure or after a certain interval of time. The process of cleaning up memory has been done for the program using a unique algorithm to reclaim memory by creating a map of reachable objects and releasing all the memory that does not have its roots in the application:

Threads						Search: <input type="text"/> <input type="button" value="X"/> Search Call Stack <input type="button" value="v"/> <input type="button" value="Gr"/>
ID	Managed ID	Category	Name	Priority		
MultipleProjects.exe (id = 5364) : C:\Users\EurotronikWin7\Documents\Visual Studio 2010\Pr						
2492	1	Main Thread	Main Thread	Normal	clr.dll!Thread::ShouldChangeAbortToUnload() - 0x5f7 bytes	
2204	0	Worker Thread	HelperCanary::ThreadProc	Normal	clr.dll!Thread::ShouldChangeAbortToUnload() - 0x53c bytes	
1824	0	Worker Thread	Thread::intermediateThreadProc	Highest	clr.dll!ManagedThreadBase_NoADTransition() + 0x35 bytes	
5260	0	Worker Thread	_TppWaiterThread@4	Normal	clr.dll!ManagedThreadBase::FinalizerBase() + 0xf bytes	
1944	0	Worker Thread	_TppWorkerThread@4	Normal	clr.dll!WKS::GCHeap::FinalizerThreadStart() + 0xb bytes	
3144	0	Worker Thread	_TppWorkerThread@4	Normal	clr.dll!Thread::intermediateThreadProc() + 0x48 bytes	
5784	0	Worker Thread	_TppWorkerThread@4	Normal	kernel32.dll!@BaseThreadInitThunk@12() + 0x12 bytes	
					ntdll.dll!__RtlUserThreadStart@8() + 0x27 bytes	
					ntdll.dll!__RtlUserThreadStart@8() + 0x1b bytes	

When running a process on Visual Studio, you can always determine the finalizer thread which is used for garbage collection from the **Threads** window. The one thread that has highest priority and has a call to `GCHeap.FinalizerThreadStart()` in its call stack is actually the finalizer thread for the process. This thread remains idle for most of the time, but occasionally, it executes and performs GC on the memory map of all threads executing on the application.



GCHeap even though it is used quite heavily, it is not exposed to the end user to call it directly. So you cannot use `GCHeap.Create` to create a new heap on your process memory and use it. Rather, it is recommended to separate each memory heap using separate `AppDomain`, which is regarded as a logical separation of memory. Each object created is associated with the current `AppDomain` class, hence when `AppDomain` is unloaded, all objects even the static variables are disposed. `AppDomain` separates the execution of code within itself. Even assemblies are loaded in `AppDomain`, such that any untrusted code cannot affect another `AppDomain` class. You can use `AppDomain.CreateDomain` to create a new `AppDomain` class to run your user code.

In this recipe we will use SOS (Son of Strike) <http://bit.ly/SOSD11> to see how these objects are getting allocated and disposed.

Getting ready

Before we actually get started with the recipe, we need to understand a few key concepts that are very important to understand, before we actually move ahead with the .NET garbage collection:

- ▶ **Stack:** The local storage is allocated on a per thread basis. When the code gets executed under CLR, the thread that is running on the code will have its local stack defined for its execution which allocates and stores local variables, method parameters, and temporary values for that particular thread. The important thing is, this memory is highly contiguous and does not employ GC to clean up data. The data for a particular method will automatically get cleared out whenever the method is returned. Another important consideration is that when each object is allocated in a managed heap, the reference is generally allocated inside the stack such that the GC tries to find the reference of the element on the thread stack to find whether the thread has this reference or not.
- ▶ **Unmanaged heap:** For a program, the unmanaged heap is used for runtime data structures, which are allocated using Win32 API calls, the MSIL, the method Tables, the JITed code, and so on. The CLR uses unmanaged heap extensively for its data structure.
- ▶ **Managed heap:** The managed heap means the memory heap that is maintained by the garbage collector. Managed objects are allocated inside managed heaps. Both allocation and deallocation of a managed heap is managed by GC and we cannot create a managed heap from an application. The `GC.Collect` API is helpful to force the GC to start allocating.

If you delve deep into how CLR allocate resources, you will see it actually maintains `NextObjPtr`, which always points to the next free space on the heap. When a process is initialized, it allocates a contiguous space (thus making it faster) on the heap for the process, and points `NextObjPtr` to the base location. As the application moves forward with memory allocation (using Win32 API `VirtualAlloc` or `VirtualAllocEx`), `NextObjPtr` moves forward to point to the next empty space that it finds. Finally, as allocation goes on and GC releases the nonreachable memory, the gaps start to appear in the heap and so, GC has to compact the heap at a certain interval. During heap compaction, the GC uses the `memcpy` function to move the objects from one memory to another to remove the unreferenced holes in memory map.

GC actually uses generations to improve its algorithm of deallocation. I have already mentioned, GC creates a map of all the objects that exists on the program assuming all as garbage and finding only the reachable memory from the program, it also marks each of the memory that is still needed by the application. This mark indicates that the memory has been moved to the next generation of GC. For the first time, every object in GC is treated as Gen 0 and after each collection, the ones that survive are marked to Gen 1 and then to Gen 2. As GC is costly, creating a memory map for the entire process memory is very costly. GC invokes its collection for Gen 0 more than what it does for Gen 1 and then to Gen 2. Hence, the short-lived objects are GCed more often than the long-lived object. GC does Gen 2 collection only when the memory pressure is very high, so the objects that have already moved to Gen 2 are not prone to garbage collection often even though it is not in use.

If we look into the sequence of GC collection:

- ▶ The EE gets suspended (**execution engine suspension**) until all managed threads have reached a safe point
- ▶ It marks all objects that do not find roots as garbage
- ▶ The GC creates a budget for each generation and determines the amount of fragmentation that can exist as a result of collection
- ▶ Deletes all objects that are marked for deletion
- ▶ Moves all reachable objects to fill the gaps (as GC heap is contiguous)
- ▶ The execution engine gets restarted

The following points continue with the key concepts you need to know about:

- ▶ **Large object heap:** It should also be noted, that the heap is classified into two types. One is **small object heap (SOH)**, which we have already stated, and another is **large object heap (LOH)**. Any object that is more than 85,000 bytes gets allocated on LOH. LOH isn't compacted, as invoking `memcpy` on such large objects is very expensive. As a result gaps may be produced for the object. In .NET 4.5, the CLR maintains a free list of dead objects so that any LOH allocation further can take up the free space rather than allocating using `NextObjPtr` of LOH.

- **Background GC:** Another important concept that you need to know is that of the background GC. Before the introduction of .NET 4.0, the .NET GC used concurrent GC to deallocate memory. The basic difference between background GC and concurrent GC is that background GC can run multiple times for a single GC generations and it uses non blocking management to GC on heaps. The basic characteristics of background GC are as follows:
 - ❑ Only full GC collection (Generation 2) can take place in the background
 - ❑ Background GC cannot be compact
 - ❑ Foreground GC (Generation 0 / Generation 1) can run parallel to the background GC
 - ❑ Full blocking GC can also happen on GC threads

Background GC increases the performance of GC and runs in parallel to the foreground GC collections.



MethodTable: This stores all information about a type. It holds information regarding static data, a table of method descriptors, pointers to `EEClass`, pointers to other methods from other VTable, and pointer to constructors.

EEClass: This holds static data information.

MethodDesc: This holds information regarding a particular method such as IL or JITed information.

How to do it...

As we already know the basic concept of how GC works, let us use SOS (Son of Strike) to debug the process to identify an object:

1. Start a console application and create a class. Let us suppose the code we wrote looks like the following one:

```
public class MyClass
{
    public static int RefCounter;
    public MyClass(string name, int age)
    {
        this.Name = name;
        this.Age = age;
        MyClass.RefCounter++;
    }
    public string Name { get; set; }
    public int Age { get; set; }
    public void GetNext(int age)
```

```

    {
        Console.WriteLine("Getting next at age" + age);
    }
}

```

Clearly, in the preceding code we use `GetNext` to get the age we pass printed on the screen and the constructor holds the value of `age` and `name`.

2. To open SOS we need unmanaged code debugging. Right-click on the project and select **Properties**. Go to **Debug** and check **Enable unmanaged code debugging**.
3. Put a call to `GetNext` in the main function. Now start debugging.
4. In the intermediate window, type `.load C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\sos.dll`, this will load the SOS for the current project.
5. Let's first see what has been produced in stack of the current thread. We use the following command:

```
!dumpstackobjects
```

This command lists all the managed objects that have been loaded into current stack. The list contains addresses and offsets from the base memory of stack.

6. To get the information about the object that we have created, take the address of the reference, which is created on stack and use the following command:

```
!DumpObj 01aebfe4
```

This will list all the necessary fields that the object has created and the value it holds with the offset. The argument that I have passed to the command represents the address of the memory location, where the object reference has been created. It also lists `MethodTable` and `EEClass` which the object uses.

7. Until now we checked the stack that has been created during the execution of the steps. Now let's look into the heap. To dump heap allocation we use:

```
!dumpheap -stat
```

This command will list all the objects that have been created on managed heap. The address that has been specified with the object is the `MethodTable` information about the objects in heap.

8. Now let's copy `MethodTable` of `MyClass` and try to see the details. We use the command:

```
!dumpheap -mt 00413910
```

This command will produce the address, the total size of the object, and the counter of objects on heap.

9. To find the GC roots for an object, copy any address of an object and use the following command:

```
!gcroot 00413910
```

This command will find the GC root address for the current object.

10. You can use `CLRStack` to get information about the entire stack trace of the currently executing assembly. With the command-line parameter `-p -1` in sequence, it produces a better result to show the parameters passed to the current method and the locals declared.

11. Copy the address of `MethodTable` information and execute the following command:

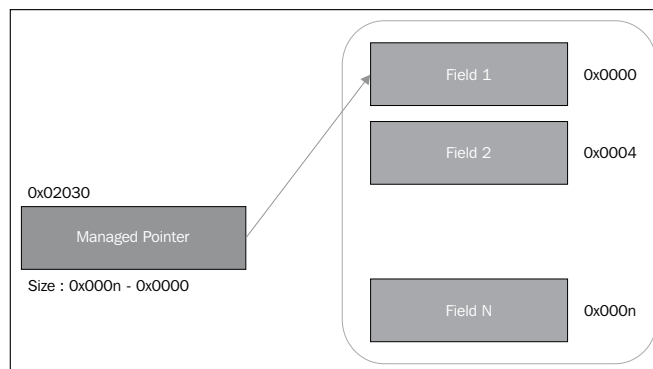
```
!dumpmt -md 0012f058
```

The output will show all the method description that is present in `MethodTable`. It also shows how the method has been compiled. For instance, pre-JIT means the assembly has already been compiled before it's executed, either using the NGen tool or using an optimization service. The JIT indicates that the method has been JITed during execution and None indicates it hasn't been JITed.

How it works...

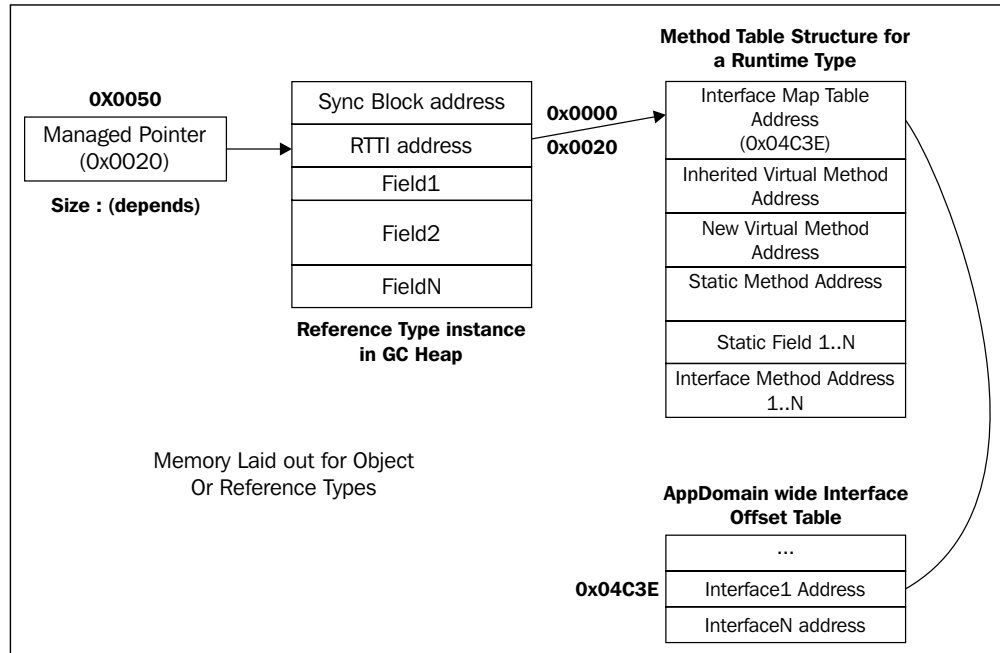
It needs to be noted that the CLR objects are either allocated into the managed heap or in a thread local stack. Even if the object remains in a heap, the reference of the object always remains in a stack, such that GC can get the information about the reference in the stack directly when GC collection is being executed. The objects in a heap allocate a memory of the `MethodTable` information, which holds information of all the methods that are present inside the object. You can easily determine the information about the location of the field using the offset value from the base address. Each heap object also contains a GC root associated with it, this indicates that the root of the object which holds the reference is not being exposed to the garbage collector.

Now, let us depict the memory allocation of a value type into memory:



The preceding diagram indicates that the managed pointer holds reference to the initial location of the actual memory. Thus in case, the managed pointer reference to 0x0000 which is the base location of Field 1 and the size of the field is 4, the next field pointer will be at 0x0004. As value types are allocated in contiguous block, we can easily use the `sizeof` operator to determine the actual size of the object.

Reference type on the contrary defines lot of additional information about the object:



In the preceding diagram, I have shown the entire layout of memory for a reference type. The initial managed pointer here for reference types holds the address of reference to **RTTI** address (**Run-time type information**). The initial 4 bytes of the memory is allocated to the synchronization block. In CLR, every object is locked inside this initial 4 bytes of memory. There is another important consideration that you need to think of, it is that every CLR object holds its type information inside it. This ensures that every object can explain its own type to itself without any dependency from outside. Hence, the reference types are self-explanatory types and programs can use this information while casting, polymorphism, dynamic binding, reflection, and so on. Even though the `MethodTable` structure resides outside the actual object, the RTTI address holds the initial address of the `MethodTable` runtime object, which holds all the information regarding the type of the object. We query the information of the runtime type using the `GetType` method from any reference type. The .NET runtime creates a special object, `Type` that helps to find out the actual type information.

There's more...

As far as we have discussed, garbage collection is a special technique implemented on CLR that automatically deals with memory management for your program. There are a number of additional properties of managed memory allocation. Let's consider few of them.

The effect of finalizer on garbage collection

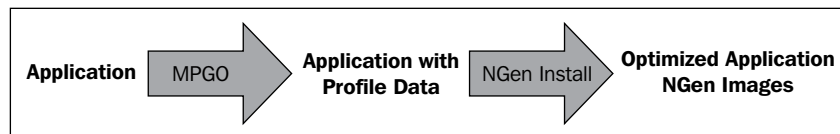
Just like any other language, .NET languages allows you to create a destructor for an object that has been created called **finalizer**. Finalizer on managed objects are not recommended. Objects that are allocated inside the heap are prone to garbage collection when there is no active reference kept for the object from the program. That means the object is of no use by a program when there is no reference of the same from the program itself.

When garbage collector is performed, it reclaims the memory of all inaccessible objects that are without finalizers and puts the objects that need to call finalize in a finalizer queue. This queue calls the `Finalize` method of all the objects sequentially and removes the objects from the list. But even though the garbage collection has already been performed on the objects and the memory has not been reclaimed by the GC, the object remains in memory until the next GC cycle executes. Hence, when you use garbage collection on a .NET object, it affects the performance of the application as the memory has not been reclaimed until GC gets executed on it for the second time (at least).

Optimizing native images using Managed Profile Guided Optimization

For a faster startup of the application, we generally precompile the assembly into cached native images using the **Native Image Generation** tool (**NGen**). With the .NET 4.5 release, there is another tool called **Managed Profile Guided Optimization (MPGO)**, which can be used to optimize the native image with even greater performance. Just like Multicore JIT, MPGO uses a profile-based optimization technology. The profile data includes scenarios or sets of scenarios, which can be used to re-order the native structures. This results in a shorter startup time and lesser working set.

The MPGO tool creates a profile for the intermediate language and adds the profile data inside the assembly as resource. NGen can later be used to precompile the IL into native code after profiling:



The profile guided optimization uses the `/LTCG:PGINSTRUMENT` compiler option newly introduced to produce profile data. A tool called `pgosweep` can be used to create a `.pgc` file that holds profile-guided information. The following command will produce a profile for the executable `myapp.exe`:

```
Pgosweep myapp.exe myapp_profile.pgc
```

Multicore JIT

JIT, or Just in time compiler, compiles the code that is written as IL and compiles back to machine code during the startup of the application. The .NET application startup is generally slower than native components as JIT needs to recompile the basic executable before it actually executes itself. Recently, CLR introduced a new feature called **Multicore Just In Time compiler**.

Talking about modern day, every PC is made up of at least two cores. Multicore JIT uses all the cores that are available to the PC and generates native code much faster than it did before. Multicore JIT shares the task into multiple cores which results in increased startup time and overall experience. The JIT creates a background thread which runs on another core to quickly JIT the code which is running. The second core runs faster than the primary and hence, compiles all the methods ahead of when it is actually needed. To know which method needs to be compiled, the feature generates profile data that is used later to determine what needs to be compiled. You can also invoke the profile data using a static method from the `System.Runtime.ProfileOptimization` class.

See also

- Visit <http://bit.ly/SOSDebug> for further reference of the SOS commands

How to find memory leaks in a .NET program

Memory leaks are a nightmares for any developer. A memory leak increases the memory of an entire application slowly, and gradually eats up the entire process memory and eventually the entire system memory. The memory leak becomes the biggest problem when the application is deployed to the server and needs to run day and night as a service.

Memory leaks can occur either in a stack, an unmanaged heap, or managed heaps. There are many ways to detect memory leaks in a program. Tools such as **Windbg**, **PerfMon**, **DebugDiag**, or even Visual Studio can be used to detect memory leaks in a program. The most important area of memory that is used by the process is represented by private bytes.



It is important to note that one should always avoid using the task manager to confirm memory leaks in a program. A task manager memory usage can be misleading most of the times because it gives information about the working set memory and not the actual memory used. Some memory of the working set is even shared between multiple processes. Hence, the task manager memory usage is not exact.

Getting ready

Memory leak in a program can create a lot of symptoms. For instance, when a program is leaking memory, it can throw `OutOfMemoryException` or it may be sluggish in responding to user input because it has started swapping virtual memory to disk or maybe the memory is gradually increasing in the task manager. When you are certain that there is a memory leak in the program, the first thing that you need to do is to detect exactly where the memory leak is occurring. In this recipe, we will try to find the memory leak of the program.

Memory leak can happen either in managed resources or in unmanaged resources. Let us try to put unmanaged resources using `Marshal.AllocHGlobal(8000)` in a timer, such that whenever the timer becomes elapsed, memory gets created.

Similarly, for managed memory leaks we use a type with one member with more than 85,000 bytes of data, so that it is created in LOH, rather than SOH. As LOH is not been compacted, it produces a large memory gap between objects which can essentially produce memory leaks:

```
public class LargeObjectHeap
{
    private byte[] buffer = new byte[90000];
}
```

This code produces a large heap allocation, and eventually invoking this multiple times will produce a memory leak in the program.

How to do it...

To deal with a memory leak, we are going to use **PerfMon**. PerfMon is an interesting tool that can be used to examine counters on process. We can detect the private bytes allocated by the process, the .NET CLR memory, the bytes on all heaps, and .NET CLR `LocksAndThreads`. Based on the counters we can determine exactly where the problem is.

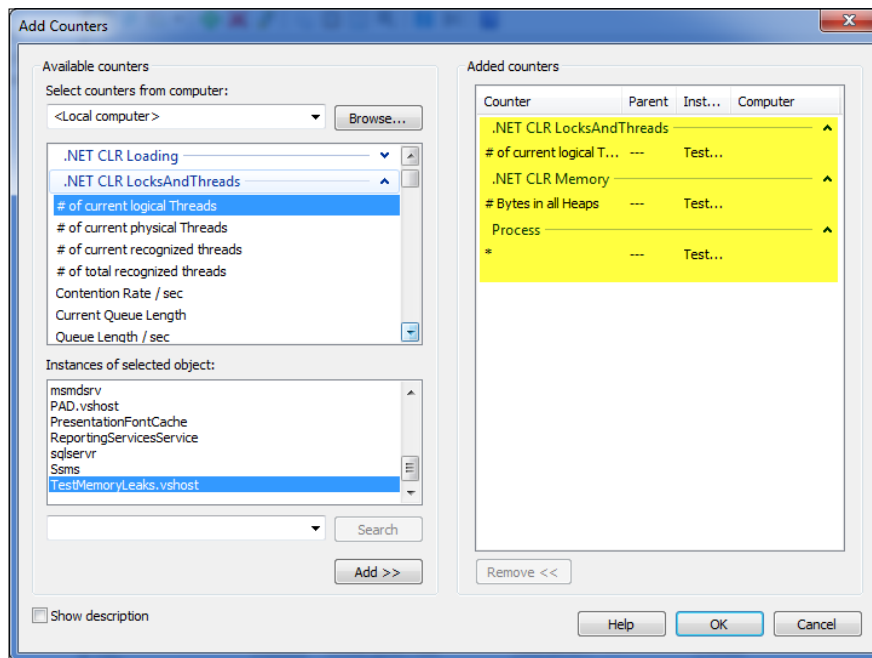
If CLR `LocksAndThreads` (**# of current logical Threads**) is increasing then the thread stack is leaking.

If only private bytes are increasing but .NET CLR memory is not increasing, then we have an unmanaged memory leak.

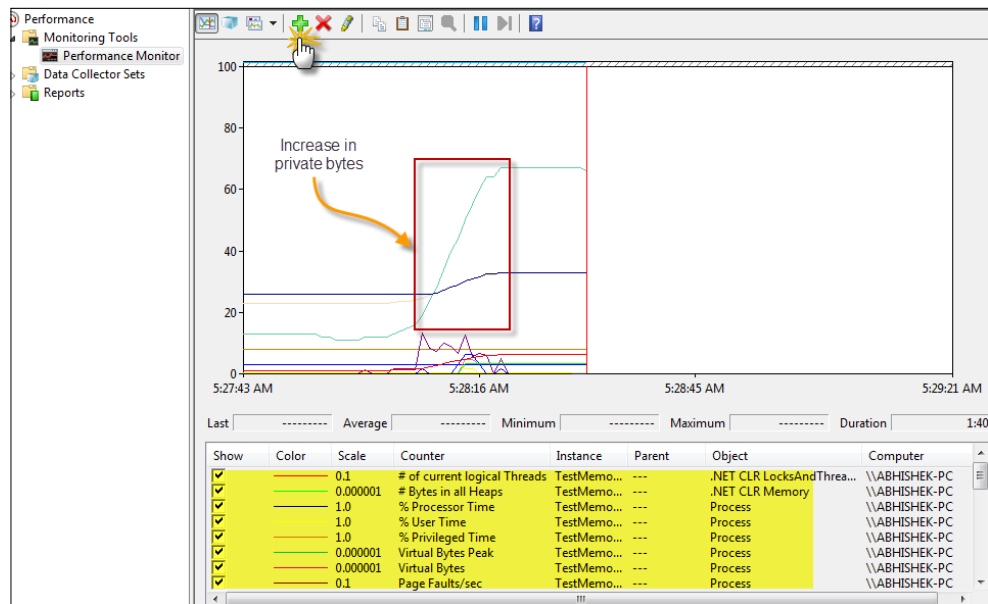
If both are increasing then it is a managed memory leak.

To work with the PerfMon tool, let's follow these steps:

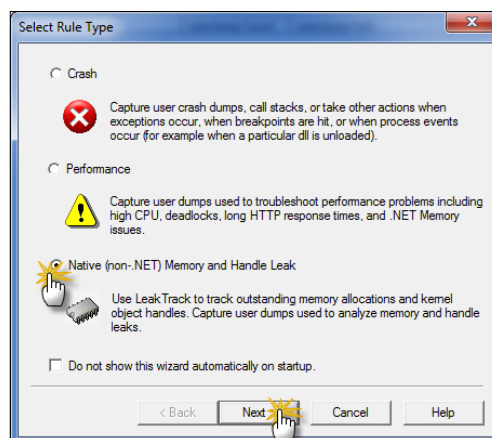
1. Start the application that has memory leaks.
2. Run PerfMon from the **Run** command to open the **Performance Monitor** tool.
3. Delete all current performance counters that are already added using the **Delete** button on top and select **Add** to add counters:



4. Select **Process** from **Performance** object, and select **Private bytes** from the list of counters. Select the appropriate process instance that is leaking memory.
5. Again, select **.NET CLR Memory** from the **Performance** object again and select **# bytes in all heaps**.
6. Finally, select **.NET CLR LocksAndThreads** in the **Performance** object and select **#of current logical Threads** from the list of counters. In the preceding figure you can see how it looks when you are adding these performance counters. I have already added the counters, so click on **OK** when complete:



7. The performance counters in the graph will show the continuous update on these counters with different colors. You can see the list of performance counters in the screenshot marked as yellow and clearly, you can identify the hike in private bytes in the picture.
8. Download the **DebugDiag** tool from the Microsoft site, start the tool and select **Native (non-.NET) Memory and Handle Leak**. The tool actually gives you three options. It allows you to check either for a crash of the application, or performance, or memory leaks. As our application probably has a memory leak, we are going to choose the one that deals with memory. You can also check performance or crash checking for your process using this tool:



9. Next, select the process that you need to check for leakage. The tool will give a list of all the processes that are running on the computer. You can select the one that is appropriate and choose **Next**.
10. Finally, select **Activate Rule** to start the monitoring.
11. After a certain amount of time has elapsed, select **Memory Pressure Analysis** from the tools and select **Start Analysis**.
12. From the HTML report you can identify how memory has been allocated in managed or unmanaged resources. It shows you a warning on the code that has been creating memory leaks.
13. Based on the report we need to go to the source code to actually fix the memory leak on the program.
14. There is no fixed rule to remove memory leak in the program, but you can use DebugDiag's analysis report to actually investigate the problem and fix it from the pointers you get from it.

How it works...

Memory leaks are generally caused by a managed application when:

- ▶ **Holding references to a managed object:** This is a situation where the variable never goes out of active scope and hence never gets exposed to GC
- ▶ **Failing to release unmanaged resources:** When dealing with unmanaged memory, loaded memory needs to be freed manually and hence releasing unmanaged memory using disposable pattern is important
- ▶ **Failing to Dispose drawing objects:** Drawing objects holds unmanaged resources, hence we need to dispose explicitly

Memory management is one of the most important considerations for any application.

There's more...

Let's talk some more about the problems and solutions that can help you to deal with memory leaks.

What are weak references in .NET

While doing actual code, sometimes it is important to know how to allocate a memory and when to create a memory location. When GC kicks in, it tries to find all the reachable types that have strong references from the application. It goes in and traces all the bits and pieces of the program, to find out if the object is somehow linked to the program, that is, it has some reference from the program that can still use this object. The GC uses this algorithm to determine whether the memory is actually garbage or not.

If we can call an object from our program, GC treats the object as "important" and marks it to move to the next generation (if not, the highest generation is reached). But it is important to remember, that when we are talking about references from the program, we only speak about strong references. A weak reference is another concept that works in contrast with the strong references.

A weak reference is an exception to GC, such that when GC kicks in, it will always thread a memory that is held by a weak reference as garbage. The garbage collection's algorithm puts a weak reference as an exception to it, and collects it when GC tries to find the reference from the program.

The .NET class library exposes a special type called `WeakReference` that actually implements the weak reference concept just stated. Sometimes it is important to create an object that can exist from the application, but when the memory pressure is high we do not want that memory to stay. In such cases, we create a weak reference and forget about the object. When we need the object again, we first try to find the object from the weak reference, if found we use it, otherwise we recreate the object again.

Hence, weak reference acts as an exception to the existing GC algorithm, which lets the GC reclaim the memory even though the memory is still accessible from the program.

Weak references can be of two types:

- ▶ **Short:** They lose the reference when GC is collected. In a general case, when we create an object of the `WeakReference` class in our program, we create this type of memory.
- ▶ **Long:** They retain their object even after the `Finalize` method has been called. The object state cannot be determined in such cases. We can pass `trackResurrection` to `true` on the constructor of `WeakReference` to create a Long weak reference:

```
WeakReference weakref = null;
private SomeBigClass _somebigobject = null;
public SomeBigClass SomeBigObject
{
    get
    {
        SomeBigClass sbo = null;
        if (weakref == null) //When it is first time or
            object weakref is collected.
        {
            sbo = new SomeBigClass();
            this.weakref = new WeakReference(sbo);
            this.OnCallBack("Object created for first time");
        }
        else if (weakref.Target == null) // when target
            object is collected
```



```
        {
            sbo = new SomeBigClass();
            weakref.Target = sbo;
            this.OnCallBack("Object is collected by GC, so
new object is created");
        }
        else // when target object is not collected.
        {
            sbo = weakref.Target as SomeBigClass;
            this.OnCallBack("Object is not yet collected, so
reusing the old object");
        }
        this._somebigobject = sbo; //gets you a strong
reference
        return this._somebigobject;
    }
    set
    {
        this._somebigobject = null;
    }
}
```

Let us suppose `SomeBigObject` is a class that creates a large amount of data inside it. So while creating an instance of that class we use the weak reference implementation of .NET and pass the object to it. We can find the object from the `Target` property only when the object has not been collected by GC. In the preceding code, you can see that we check whether the object has already been collected or not and depending on the same we create the object again or pass the existing object.

The `Target` property gets you a strong reference from `WeakReference`. When you are holding the value of `Target` to some reference in your program, the GC will not collect the object. But as soon as the reference is nulled and no reference to the `Target` object has been stored, the object inside `WeakReference` is exposed for collection.

Here in the preceding code, we have also created a callback just to print the message to the caller whether the object is recreated or not.

What are lazy objects in .NET

In our previous code we see how we can create a weak reference that can be used while dealing with very big objects. The concept of lazy objects on the other hand is a pattern that defers the initialization of the object when it is really required. A lazy object in .NET actually allows you to specify a callback that will automatically be called whenever the object is invoked. The idea is to pass the callback which creates the object, in such a way that when the object actually needs to be created, the lazy object will invoke the handler and create the object.

There are three types in .NET that support lazy Initialization:

- ▶ **Lazy:** This is just a wrapper class that supports lazy initialization
- ▶ **ThreadLocal:** This is the same as `Lazy` but the only difference is that it stores data on thread local basis
- ▶ **LazyInitializer:** This provides static implementation of lazy initializer that eliminates the overhead of creation of lazy objects

Lazy

`Lazy` creates a thread-safe lazy initialization of objects. Say for instance, when we need to create a large number of objects and each of these objects by itself create a lots of objects, it is easy to initialize the master object using `Lazy` blocks. For instance, say there are a large number of customers and for each customer, there are a large number of payments, if `Customer` is an entity and `Payment` is also an entity, `Customer` will contain an array of `Payment` objects. Thus, each entity requires a large number of database calls to ensure that the data is retrieved, which doesn't makes sense. Using the `Lazy` class you can eliminate this problem.

Let us look at how to use it:

```
public class Customer
{
    public string Name { get; set; }
    public Lazy<IList<Payment>> Payments
    {
        get
        {
            return new Lazy<IList<Payment>>(() => this.
FetchPayments());
        }
    }

    private IList<Payment> FetchPayments()
    {
        List<Payment> payments = new List<Payment>();
        payments.Add(new Payment { BillNumber = 1, BillDate =
DateTime.Now, PaymentAmount = 200 });
        payments.Add(new Payment { BillNumber = 2, BillDate =
DateTime.Now.AddDays(-1), PaymentAmount = 540 });
        payments.Add(new Payment { BillNumber = 3, BillDate =
DateTime.Now.AddDays(-2), PaymentAmount = 700 });
        payments.Add(new Payment { BillNumber = 4, BillDate =
DateTime.Now, PaymentAmount = 500 });
        //Load all the payments here from database
        return payments;
    }
}
```

```

    }
    public Payment GetPayment(int billno)
    {
        if (this.Orders.IsValueCreated)
        {
            var payments = this.Payments.Value;
            Payment p = payments.FirstOrDefault(pay => pay.
            BillNumber.Equals(billno));
            return p;
        }
        else
            throw new NotImplementedException("Object is not
            initialized");
    }
}

public class Payment
{
    public int BillNumber {get;set;}
    public DateTime BillDate { get; set; }
    public double PaymentAmount { get; set; }
}

```

Here I have created a class called `Payment` that has a few properties. Each customer has a list of payments. You can see the `Customer` class that uses Lazy implementation. This calls `FetchPayments` to create objects whenever `this.Payments.Value` is called. The `IsValueCreated` property will evaluate to `true` when the list is created.

Similar to the list, you can use Lazy binding to any other CLR objects as well.



`System.Lazy` creates a `ThreadSafe` object by default. The default constructor creates an object with `LazyThreadSafetyMode.ExecutionAndPublication`. Thus, once an object is created by one thread, the object will be accessible to all other concurrent threads.

ThreadLocal

Similar to Lazy, `ThreadLocal` creates an object local to one thread, so each individual thread will have its own Lazy initializer object and hence, will create the object multiple times once for each thread. You can create objects that are local to one thread using the `ThreadStatic` attribute. But sometimes `ThreadStatic` fails to create a true `ThreadLocal` object. Basic static initializer is initialized for once, in case of the `ThreadStatic` class.

ThreadLocal creates a wrapper of Lazy and creates a true ThreadLocal object:

```
public void CreateThreadLocal()
{
    ThreadLocal<List<float>> local = new
ThreadLocal<List<float>>(() => this.GetNumberList(Thread.
CurrentThread.ManagedThreadId));
    Thread.Sleep(5000);
    List<float> numbers = local.Value;
    foreach (float num in numbers)
        Console.WriteLine(num);
}

private List<float> GetNumberList(int p)
{
    Random rand = new Random(p);
    List<float> items = new List<float>();
    for(int i = 0; i<10;i++)
        items.Add(rand.Next());
    return items;
}
```

In the preceding methods, CreateThreadLocal creates a local thread and takes the lazy object GetNumberList when the value is called for (just like normal lazy implementation).

Now, if you call CreateThreadLocal using the following code:

```
Thread newThread = new Thread(new ThreadStart(this.
CreateThreadLocal));
    newThread.Start();
    Thread newThread2 = new Thread(new ThreadStart(this.
CreateThreadLocal));
    newThread2.Start();
```

Each thread newThread and newThread2 will contain its own list of lists.

LazyInitializer

Finally, coming to LazyInitializer, you can create the same implementation of the lazy objects without creating the object of Lazy. LazyInitializer handles the lazy implementation internally giving you static interfaces from outside that enable you to use it without much heck.

The `LazyInitializer.EnsureInitialized` method takes two arguments, in general. The first one is the `ref` parameter, where you have to pass the value of the variable. Here, you want the target to be generated and the delegate that generates the output:

```
public void MyLazyInitializer()
{
    List<Payment> items = new List<Payment>();
    for (int i = 0; i < 10; i++)
    {
        Payment paymentobj = new Payment();
        LazyInitializer.EnsureInitialized<Payment>(ref paymentobj, () =>
        {
            return this.GetPayment(i);
        });
        items.Add(paymentobj);
    }
}
```

In the preceding scenario, you can see that I have used `LazyInitializer` that fetches each `Payment` object when it is required. The `ref` parameter takes a class type object explicitly and returns the object to the variable.



When should we use Lazy?

If you are using an object base which is resource consuming and you are sure that every object will not be required for the application to run, you can go for lazy initializers, otherwise it is not recommended. Another important consideration is that `Lazy` doesn't work very well with value types, and it is better to avoid it for these.

How to use Visual Studio to create memory dump files

Visual Studio allows you to store and create dump files during debugging sessions. These dump files allow you to save the entire dump information of a program when a program crashes. With the use of dump files, you can debug it later whether on a build computer or another computer that has the source code and debugging symbols.

Visual Studio 2012 debugger can save mini dump files for either managed or native code. To create a dump, select **Debug | Save Dump file**. You can either select **Minidump** or **Minidump with heap**.

Once the dump has been saved, you can use this dump to open in Visual Studio from **File | Open** to get the entire report of the dump. From the **Action** section, you can debug the program either with native only or with mixed.

How to isolate code using AppDomain

In .NET, the primary execution unit of an application is not a process but rather an AppDomain. AppDomain is a separate unit that exists in the .NET environment that loads memory and runs user code. Such that the memory that has been allocated in one AppDomain instance is totally isolated from another one. Unlike threads, an AppDomain instance does not share the same memory, and hence any corruption of memory in one domain cannot affect the other.

While doing memory management of an application, it is good to use separate AppDomain instances to load the assembly that leaks memory. This will ensure that the assembly executes in a separate and isolated memory.

You can use the following code to create an AppDomain instance:

```
AppDomain newDomain = AppDomain.CreateDomain("domainName");  
newDomain.ExecuteAssembly(yourassembly);
```

Even though we do not create an AppDomain instance during the execution of any application, the CLR creates a default AppDomain instance for us. You can use `AppDomain.CurrentDomain` to get information about the domain where the user code is executing.

Solutions to 10 common mistakes made by developers while writing code

Being a developer has always been a mammoth task to handle bad code. Most developers need to work in teams such that one's code needs to be recompiled and used by others. Mistakes in development by one of them adversely affects the entire application. Even most current programmers do not know the mistakes that they are making in the application. In this recipe, I am going to cover some of the interesting common mistakes that I see while working with developers and discuss their solutions.

Getting ready

We create a console application to demonstrate each case.

How to do it...

In this recipe, let's add some of the interesting mistakes which developers do, often either unknowingly or mistakenly. We will discuss each of the points individually:

1. Use of a destructor instead of `IDisposable`
2. Forgetting to call `Dispose` before going out of scope
3. Forgetting that strings are immutable

4. Breaking the call stack of an exception by throwing the exception again
5. Calling object members without making sure that the object cannot be null
6. Forgetting to unhook event handlers appropriately after wiring them
7. Not overriding `GetHashCode` when overriding `Equals`
8. Forgetting the call to the base constructor when calling derived objects
9. Putting large objects into static variables
10. Calling `GC.Collect` unnecessarily

How it works...

Let's see how we can solve the problems we described.

- ▶ Use of a destructor instead of `IDisposable`

If you are declaring a class which uses references to unmanaged resources, it is important to dispose the object when it is going out of scope. In .NET you can use a destructor as its predecessor to release unmanaged resources which include file handles, database handles, and so on, such that the .NET environment calls the destructor automatically whenever GC gets executed. But this approach even though you don't need your caller to be aware of anything, is non-deterministic and adds some expenses. First of all, GC is non-deterministic, and hence the caller cannot make sure that the object has been disposed only after the object is dereferenced. The caller needs to wait for the GC to execute its finalizer until the resources are free from the object. Another important drawback is that the destructor loses the GC cycle, and hence the object will remain in memory for a longer time than expected. So if the class holds a reference to a large object, it might cause a delay in reclaiming the memory:

```
public class FinalizerClass
{
    public object LargeObject { get; set; }
    ~FinalizerClass()
    {
        //Release unmanaged references
        this.LargeObject = null;
    }
}
```

In the preceding code, `LargeObject` is disposed from the finalizer thread, hence once the object is sent out of scope, it has to wait for GC to execute the destructor.

.NET comes with a better approach with the disposable pattern, allowing the caller to release its resources. Thus, the caller knows when to release the resources and this approach works in a deterministic manner:

```
public class DisposableClass : IDisposable
{
    public object LargeObject { get; set; }
    public void Dispose()
    {
        //Release unmanaged references
        this.LargeObject = null;
    }
}
```

Here, the caller needs to call `Dispose` whenever it wants to release the resources from memory. The disposable pattern does not lose the GC cycle and hence works best in a managed environment.

► Forgetting to call `Dispose` before going out of scope

Just similar to previous example, when you are using a disposable object, it is important to call the `Dispose` method before the object goes out of scope to release unnecessary resources from memory. C# comes with a shortcut to ensure that the object calls the `Dispose` method when going out of scope. It is always a good idea to use the `using` block when using a disposable object to avoid the problem:

```
public void CallDisposable()
{
    using (DisposableClass dclass = new DisposableClass())
    {
        //Working with the dclass
    }
}
```

In the preceding code, the `Dispose` method will automatically be called when `dclass` is going out of scope of the `using` block.

► Forgetting that strings are immutable

In .NET, strings are immutable. So when you do a slight manipulation on a `String` element, the entire memory is recreated again to store the final result. The value of a string once created cannot be changed. For instance:

```
public void StringWhoes()
{
    string thisString = "This " + "is " + "a " + "string.";

    thisString.Replace("is", "was");
}
```



```
Console.Write(thisString); //wrong

thisString = thisString.Replace("is", "was");

Console.Write(thisString);
}
```

In the preceding code, we declared a string dynamically. It is important to note that (even though here the IL optimization does not affect any performance) strings are immutable, hence for each string concatenation, a new memory is created. It is better to use `StringBuilder` or `string.Format` rather than `String` when we need to append strings with some other variables. Another interesting fact is that as strings are immutable, you cannot replace the object string using the `Replace` method. You need to store the result into another reference to get the result from `Replace`.

- Breaking the call stack of an exception by throwing the exception again

Breaking the call stack of an exception is another important mistake that developers often make. Let us look at the following code:

```
public void BreakCallStack()
{
    try
    {
        this.Call1();
    }
    catch (Exception ex)
    {
        throw ex;          // Wrong
        throw new ApplicationException("Exception occurred", ex);
    }
    // wrong

    throw;                // right
}

private void Call1()
{
    this.Call2();
}

private void Call2()
{
    throw new NotImplementedException();
}
```

In the preceding code, `BreakCallStack` calls a method `Call11` which in turn calls `Call12`. Now when the exception is caught in `BreakCallStack`, it will hold the information of the entire stack in `ex`. Calling `throw ex`, or wrapping the exception into another exception will eventually lose the stack. So rather than using the first two constructs it is good to use the third construct (`throw`) to retain the call stack.

- ▶ Calling object members without making sure that the object cannot be null

`NullReferenceException` is one of the most common forms of exception that we see regularly. While developing code, it is always important to check every parameter that is coming as an argument, or any external object to null before calling its members:

```
public string CallWithNull(object param1)
{
    return param1.ToString();
}
```

In the preceding code the method is called with a parameter `param1` of which we pass the `ToString` implementation. Now if the `param1` parameter holds null during any call, the method will produce `NullReferenceException`. Hence, it is important to check if `param1` equals null before invoking any method.

- ▶ Forgetting to unhook event handlers appropriately after wiring them

A memory leak can occur when an event is hooked to an event source but never unhooked. Generally, an event takes a delegate as argument, which is passed from the caller. But if you do not unhook the event handler from the actual object where the event exists, the object will not be garbage collected until the object that holds the event handler gets exposed to GC. Events in .NET holds a strong reference to `EventSource`, and hence it is very important to unhook the event handlers when not in use.

- ▶ Not overriding `GetHashCode` when overriding `Equals`

Sometimes developers forget to override `GetHashCode` even after the overriding `Equals`. When we equate two objects, the .NET first calls `GetHashCode` of each of the objects before calling the `Equals` method. So if the hash code of two objects does not match, they will never be considered as equal. Hence, you need to override `GetHashCode` of both the types and return the same value for two objects.

- ▶ Forgetting the call to the base constructor when calling derived objects

Sometimes we forget to call the base object from the constructor of a derived object. It is important to note that we must construct the base object totally before doing the work in the derived object. For instance:

```
public class DerivedNode : BaseNode
{
    public DerivedNode(string derivednote) //wrong
    {
```

```
        this.DerivedNote = derivednote;
    }
    public DerivedNode(string basenote, string derivednote) //
right
        : base(basenote)
    {
        this.DerivedNote = derivednote;
    }

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

In the preceding code, the first constructor does not call the base constructor but rather calls the default constructor that does nothing, whereas the second constructor actually creates the base object completely.

- Putting large objects into static variables

Static variables are disposed once `AppDomain` is unloaded. Hence, when you are using a static object, you need to remember that those are long living objects and will never be reclaimed during the execution of the process until `AppDomain` is unloaded (which is a rare scenario for a program). Hence, putting a large object in a static variable will put additional pressure on the process memory. Another important consideration that you need to remember is that calling a static member from multiple threads can create a nightmare. You must make static members thread safe when being accessed from concurrent threads.

- Calling `GC.Collect` unnecessarily

Developers are often prone to call `GC.Collect` to invoke the GC cycle. Generally, GC collection blocks the execution engine if it is not a background GC and hence, can create performance bottlenecks in the application. GC collection is automatically managed by the CLR, and hence it is recommended to avoid the explicit call.

There's more...

Even though we have already discussed the 10 common mistakes that developers often do while developing an application, it is not an entire list. Let's identify some of the additional mistakes that can be taken care of.

Never declare structs if you don't have good reason to

I have seen developers declare structs in their code when they do not need member functions or inheritance. Generally, it is important to note that `struct` even though it exists in languages, should be avoided. Generally, the limit of `struct` is 16 bytes, if it is anything over that threshold, the performance of `struct` degrades from a class and it is better to use a class instead of `struct`.

Another interesting fact is that, a struct will automatically allocate itself whenever the object is declared, and you cannot put your custom logic into the default constructor of a struct. Structs should always be immutable if it is declared.

Do not create a list from IEnumerable before iterating on it

`IEnumerable` is a special interface that implements a state machine to generate the sequence of objects. It is important to note that sometimes when we need to directly call one object with its index, you need to create a deterministic list of objects from the `IEnumerable` list. But you should always remember, `IEnumerable` is good at iterating. So, if you do not need a cached implementation, and just need to use the list to iterate on an object, it is not good to convert `IEnumerable` to a list.

For instance, consider the following code:

```
public void GetList(IEnumerable<string> strings)
{
    List<string> lststrings = strings.ToList();
    foreach (string lstring in lststrings)
    {
        //Write custom logic
    }
}
```

In the preceding code, it is not necessary to create a list of strings before going to the `foreach` loop. The `ToList` extension method actually generates the list by iterating using `foreach` and putting it into the cached list. So here, the performance is degraded because the same sequence is iterated twice.

See also...

- ▶ See <http://bit.ly/ProgrammingMistakes>

3

Asynchronous Programming in .NET

In this chapter, we will cover the following recipes:

- ▶ Introduction to Threading and Asynchronous Threading patterns
- ▶ Working with Event-based asynchronous pattern and BackgroundWorker
- ▶ Working with Thread locking and synchronization
- ▶ Lock statement using task-based parallelism in concurrent programming
- ▶ Working with async and await patterns
- ▶ Working with Task Parallelism Library data flows

Introduction

We are in a world of continuous development. The more we move towards the future, the more we see the advancements in technology around us. The hardware that we use today is becoming more and more improved. Today, almost all the desktops have at least two cores installed. The power of the CPU is also constantly increasing day-by-day.

As a matter of fact, with the improvements in technology and hardware the bar of expectation towards an application has also increased considerably. The issues and requirements that we would have earlier neglected becomes a prominent need for any application built in today's world. There has already been a revolution with User Interfaces. We now use rich graphics to develop user interfaces employing superior graphics drivers for professional applications. As our hardware supports multiple cores, as an application developer, it is important to utilize all of the cores to get output quickly. To run a program in multiple cores, we need Threading patterns to share the complex work into multiple parallel executing threads without blocking the UI threads. So as we are moving further in the future, the application developer needs to adapt the Threading pattern frequently and use it in the primary development of any application.

The Asynchronous programming approach is a way to enable developers to use either threads or do something that does not block an existing sequence of the program. Any modern day application strongly recommends this feature, even though the latest trend in technology prohibits you from writing applications that do everything in synchronous mode and block the user from doing anything until the ongoing process finishes its execution. We will learn more about Metro apps and/or Windows Phone apps which do not allow you to even block your UI threads for more than a few seconds. So the idea of asynchronous programming is very common in modern day programming. All the major languages are trying to give most APIs to the developers, so that they can write the asynchronous code more efficiently and easily.

Writing an asynchronous code is not easy. There are lots of things that you need to consider while transforming a sequential code into an asynchronous mode. When there is an inclusion of threads, we need to consider a lot of different things like thread synchronization, interlocking between threads, concurrency, shared resources, thread local storages, and so on. All these things wouldn't be required if you write the same code sequentially.

As an application developer, you also need to go through a lot of complexity when you are developing an application that requires multi-threading, rather than an application that does not require it. As the complexity increases, it becomes difficult for an ordinary programmer to understand the logic easily and hence the learning curve increases. The application programmer also needs to ensure when they actually require the introduction of threads, and how many threads he/she should be creating for an application he/she is building. He/she would also need to consider when thread locking is required and when it isn't. So there are lots of strategies that an application developer needs to go through while developing an application.

.NET languages are creating new patterns to support the flexibility of the user to go with threads easily and seamlessly. It introduces **Begin** and **End** patterns which ensure that even though the code is running on the same thread, it still does not block the thread when some external resources are executed. Recently C# introduced `async` and `await` patterns that made the life of a .NET developer even easier. This pattern lets the user define the asynchronous method in the same way as he/she would define a synchronous block. Just specifying or annotating the method with a `sync` keyword ensures that the compiler rewrites the block completely to orchestrate the execution of the asynchronous block.

.NET has made changes to its components which enable the programmer to manage threads easily reducing the overall operating system thread on a process when the application goes on creating orphan threads. Most of the complexities are automatically managed by it. In this recipe we will consider the use case scenarios of asynchronous operations with examples.

Introduction to Threading and Asynchronous Threading patterns

Threads have been around for a long time. It is said that to support concurrency, operating systems need to create threads and run two blocks of code in parallel. So when we create threads, we create one more execution context and when we start a thread the operating system automatically schedules a thread which will run in parallel while executing the program. Threads ensure that while executing one operation, the other operation remains either idle or shares the CPU, executing its own code in parallel.

Let us consider the following code:

```
static void Main(string[] args)
{
    Thread thNew = new Thread(WriteThreadName);
    thNew.Name = "Worker Thread";

    Thread.CurrentThread.Name = "Main Thread";

    thNew.Start();

    WriteThreadName();

    if(thNew.IsAlive)
        Console.WriteLine("Thread is still alive");

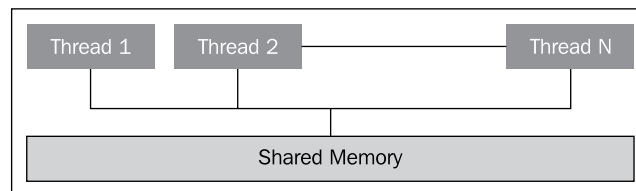
    Console.ReadKey(true);
}

public static void WriteThreadName()
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(Thread.CurrentThread.Name);
    }
    Console.WriteLine("Thread is now closing");
}
```


Thread is the .NET class that creates a new thread. We need to pass a delegate to the thread constructor which is either `void` or takes an `object` argument. In our case, `WriteThreadName` is the method that is passed as a delegate to the thread object. Each thread has `Name` which identifies the current thread and by calling `Start` on that `Thread`, the execution of the method gets started. The previous code prints the name of the thread 10 times, and prints another message before exiting the method. You should note that we have called `WriteThreadName` from the `Main` method too, which is called sequentially and will print the thread name sequentially. `Thread.CurrentThread` gets the currently executing thread, and hence we name the `Main` thread using `Thread.CurrentThread.Name`. If you look at the output it appears as follows:

```
Worker Thread
Worker Thread
Worker Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Worker Thread
Main Thread
Main Thread
Main Thread
Thread is now closing
Thread is still alive
Worker Thread
Worker Thread
Worker Thread
Worker Thread
Worker Thread
Worker Thread
Thread is now closing
```

Clearly, there is an overlap of `Worker Thread` and `Main Thread`. If the `Thread.Start` method is called sequentially, all the `Worker Thread` methods will be printed first and then `MainThread` gets printed. But as the two threads run in parallel, the overlapping of the thread occurs based on the CPU time each execution thread gets. The `IsAlive` property determines whether the thread has finished its execution or it is still executing. Here you can see the worker thread is still active even after the main thread has finished execution.



A process is created with a number of threads which share memory between them. Thread is the unit of processing, so if there are five threads in a process, the process will share five timeslices for each execution context and share the CPU cycles together. Each thread has its own memory associated with it called thread local memory, that associates a Stack which is the main memory for the executable value type allocation and references, its context, and so on. All the threads in the process share the memory that has been allocated by the process.



In an ideal situation there should be as many threads and as many processors installed in the machine. Or more precisely, there should exist one thread per processor to avoid context switch overhead on multiple threads running on same processor.

For a particular thread there are a few methods that have special significance, as follows:

- ▶ **Start:** Invoked to Start a Thread
- ▶ **Suspend:** Suspend a thread from execution
- ▶ **Resume:** Suspended Thread can be resumed
- ▶ **Abort:** Aborting a thread means cancelling the execution of the Thread completely
- ▶ **Sleep:** Suspend the thread execution for a certain milliseconds specified as argument
- ▶ **Join:** The current thread is blocked until the thread for which the Join is called has finished execution

There is also the concept of `ThreadPool` which allows you to manage the Threads. We can queue work items on a `ThreadPool` using `ThreadPool.QueueUserWorkItem` to actually queue a method for execution. We can also configure the `ThreadPool` Maximum and Minimum threads which get the queued work items and executes them. It is important for a program to always use `ThreadPool` to ensure that the program does not fall into large number of threads situation.

Each thread in .NET consumes 4 MB of memory, and hence creating threads in .NET process or anywhere is expensive.

.NET asynchronous programming comes with mainly three patterns:

- ▶ Asynchronous Programming Model (APM)
- ▶ Event-based Asynchronous Pattern
- ▶ Task-based Asynchronous Pattern (TPL)

Each of these models of asynchronous patterns gradually made it easier to use asynchrony in an application. In this recipe we will cover the Asynchronous Programming model, and show you how you can use it in your application to implement asynchrony.

How to do it...

To understand asynchronous patterns further let us go ahead and follow the steps to create an application:

1. To use Asynchronous pattern, let's create a method that needs to be called through this model. We call it `CallNormalAsyncMethod`.
2. Each Async pattern is associated with respective Begin and End patterns, where the Begin pattern actually indicates the start of the call, and the End pattern indicates the close of the method. For instance:

```
public void CallNormalAsyncMethod()
{
    Thread.CurrentThread.Name = "Worker Thread1";
    Func<int, int> faction = this.GetSquare;
    faction.Invoke(2); // Synchronous call

    faction.BeginInvoke(2, this.GetSquareCall back,
"First");

    Console.ReadKey(true);
}

private void GetSquareCall back(IAsyncResult result)
{
    if (string.IsNullOrEmpty(Thread.CurrentThread.Name))
        Console.WriteLine("Its a New Thread");

    AsyncResult arestult = result as AsyncResult;
    Func<int, int> fdelegate = arestult.AsyncDelegate as
Func<int, int>;

    if(arestult.IsCompleted)
        Console.WriteLine("EndInvoke need to be called,
{0}", arestult.AsyncState);

    int retVal = fdelegate.EndInvoke(arestult);

    Console.WriteLine(retVal);
}
```

3. The code in step 2 uses the `BeginInvoke` and `EndInvoke` patterns. We assign the method to a delegate of same type and call its `BeginInvoke`. We know that delegates can be called using the `Invoke` method to run sequentially, while it also supports two more methods, `BeginInvoke` and `EndInvoke`, which actually runs the method in a separate thread.

4. Each `BeginInvoke` call takes the arguments based on the delegate, a call back, and a State object. In the State object you can pass anything which you are going to propagate with the method. There is also a Call back that you need to pass as an argument. Call back is of the `AsyncCall back` type.
5. Call back gives an object of `AsyncResult` as argument. This object holds the total information about the whole operation. For instance, we can retrieve the actual delegate from `AsyncDelegate` property; you can also use the `IsCompleted` property to find whether the call has finished execution. There is another special property called `CompletedSynchronously`. Actually `BeginInvoke` and `EndInvoke` do not indicate only asynchronous operation. Sometimes it can also work synchronously. It will be evaluated to true if the operation is synchronous. You can test this by calling `EndInvoke` directly after the `BeginInvoke` statement.
6. The `BeginXXX` and `EndXXX` pattern has been also added to the Base Class Libraries. For instance, `FileStream` has a `BeginRead` and `EndRead` method respectively that works exactly the same way as it does with `BeginInvoke` and `EndInvoke` patterns. Let us look at the following code:

```
public void ReadFile(string filename)
{
    if (File.Exists(filename))
    {
        FileStream fstream = new FileStream(filename, FileMode.
Open);
        byte[] buffer = new byte[1000];
        AsyncCall back rfCall back = this.ReadFileCall back;
        IAsyncResult r = fstream.BeginRead(buffer, 0, buffer.
Length, rfCall back, fstream);
    }
}

private void ReadFileCall back(IAsyncResult result)
{
    FileStream fs = (FileStream)result.AsyncState;
    int bytesRead = fs.EndRead(result);

    Console.WriteLine("Bytes read {0}", bytesRead);
    fs.Close();

    Console.ReadKey(false);
}
```

Here `BeginRead` invokes the read operation asynchronously, and hence after the file has been read, it will call `Call back` from which you can call `EndRead` to get the number of bytes read as the result.

How it works...

The `BeginXXX` and `EndXXX` patterns were introduced with .NET 1.1 which allow you to create a non-blocking call to a method that may or may not include a **Thread**. It is very important to remember that when `BeginXXX` and `EndXXX` involve anything other than CPU intensive work, it does not create a new **Thread**. For instance, while reading a disk file, we can use the `BeginRead` and `EndRead` patterns which reads data from `FileStream` and do not block the caller UI Thread. The `Begin/End` patterns are not Thread locking.

The asynchronous pattern creates a new **Thread** from the `ThreadPool` and assigns that thread to the current work in the queue. If the maximum thread has already been reached, no more worker thread will be created for the current execution, but will queue the object to the `ThreadPool`.

To understand this better we need to take a detailed look at the types: `IAsyncResult` and the `AsyncCallback`.

The `AsyncResult` interface defines the status of the asynchronous operation. It sets the properties which directly maps to the operating system threads and thus gives the entire information about the status of the execution context. `AsyncResult` has few properties that need attention:

- ▶ `AsyncState`: This is the state object that is propagated throughout the asynchronous operation.
- ▶ `AsyncWaitHandle`: This is the `WaitHandle` instance that can be used for waiting for the asynchronous operation to complete.
- ▶ `CompletedSynchronously`: This indicates whether this is a synchronous or asynchronous call.
- ▶ `IsCompleted`: This indicates whether the call is completed or not. This is a non-blocking call and immediately returns the completion state of the operation.

The `AsyncCallback` automatically gets `IAsyncResult` as the argument that can be used to invoke the call back to the operation when complete.



It is important to note that this pattern does not support Exception handling, Cancellation, and Progress monitoring explicitly.

There's more...

Threading and its patterns have a few additional things to be learned. Let us discuss them to complete the recipe.

Relation between a Process, AppDomain, and Thread

A process is in simple terms an executing program. It is a unit of application which can independently run on the operating system. A process allocates its own memory area that needs to be used by the application context. The application requests memory through the process.

Application domains are special logical containers of a CLR program which separate the Threads into isolation. In other words, in .NET there is an abstraction level which manages its own memory and separates the memory and execution context in logical separation called `AppDomain`. .NET automatically creates a new `AppDomain` by default when the application is launched.

Any static memory or threads are allocated inside an `AppDomain` such that when the `AppDomain` is unloaded, all the memory associated with the `AppDomain` ceases to exist and aborts execution. `AppDomain` provides a mechanism to create a logical separation of threads running on the application and makes you create an isolation.

Threads are the units of execution as we already know. Each thread in .NET is associated with an `AppDomain` which has access to shared resources and memory.

What is a Waithandle class and why is it important?

`WaitHandle` is used for signaling. Sometimes it is important to stop execution of one thread until a portion of another thread has executed. To receive notifications we use `WaitHandle`, we wait for a signal from one thread and set the signal on another. The `WaitHandle` class is of three types:

- ▶ `AutoResetEvent` works like a Ticket turnstile. You can block a thread using `WaitOne`, which opens one turnstile until another thread unblocks it using the `Set` method. If more than one thread uses `WaitOne`, it maintains a queue of turnstile.
- ▶ `ManualResetEvent` functions like an ordinary gate allowing any number of threads to pass through. Calling `Set` opens the gate allowing the `WaitOne` to let through. `Reset` closes the gate and after that any `WaitOne` class will block the thread until it is opened again. .NET recommends using `ManualResetEventSlim` (which is available from .NET Framework 4.0 and onwards) rather than `ManualResetEvent`, as it is optimized and allows you to pass `CancellationToken.L`.
- ▶ `CountdownEvent` lets you wait for one or more threads until the counter is reached. You can signal a thread using the `Signal()` method but it will block until the number of signal mentioned on the counter is encountered.

```
static AutoResetEvent autoset = new AutoResetEvent(false);
static ManualResetEventSlim mreset = new ManualResetEventSlim(false);
static CountdownEvent creset = new CountdownEvent(3);

static void Main(string[] args)
{
```

```
Thread th = new Thread(CallThread);
th.Start();
Console.WriteLine("auto blocking");
autoreset.WaitOne();
Console.WriteLine("auto Unblocked");

Console.WriteLine("manual blocking");
mreset.Wait();
Console.WriteLine("manual unblocked");

Console.WriteLine("countdown blocking");
creset.Wait();
Console.WriteLine("countdown unblocked");
Console.ReadKey(false);
}

static void CallThread()
{
    Console.WriteLine("thread Called");
    Thread.Sleep(1000);
    autoreset.Set();
    Console.WriteLine("thread signalled auto");
    mreset.Set();//unblocks all Waits.

    //mreset.Reset(); //blocks again for all wait call

    creset.Signal();
    creset.Signal();
    creset.Signal(); // Need 3 calls to unblock
    Console.WriteLine("Countdown reset");
}
```

In the given code demonstration, `AutoResetEvent` works just with `WaitOne` and `Set`. `ManualResetEvent` blocks until `Set` is called and before `Reset` is again called. `CountdownEvent` works when the number of signals reaches a certain counter.

How does the Barrier class in .NET work?

`Barrier` is a new type introduced in .NET 4.0. It allows the user to define the synchronization primitives and lets your threads run code simultaneously in predefined phases.

A `Barrier` class defines phases in such a way that the thread executing a set of code reaches a certain point where it is stopped and sent to wait until all the threads running in `Barrier` get signal. The set of code that it executes is called pre-phase work until each of them reaches a `Barrier`. Once all the threads reach `Barrier`, it executes the post-phase code.

Thus the Barrier class is used in very specific cases, wherein you are required to specify a barrier for the threads to stop and wait for other threads to finish the execution, and finally run a method specified as call back to the Barrier as post-phase code.

Let's look for the same in the following code:

```
public static Barrier barrier = new Barrier(5, e =>
{
    Console.WriteLine("All threads Finishes");
});
static void Main(string[] args)
{
    Thread t1 = new Thread(() => ProcessPhaze1());
    Thread t2 = new Thread(() => ProcessPhaze1());
    Thread t3 = new Thread(() => ProcessPhaze1());
    Thread t4 = new Thread(() => ProcessPhaze1());
    Thread t5 = new Thread(() => ProcessPhaze1());

    t1.Start();
    t2.Start();
    t3.Start();
    t4.Start();
    t5.Start();

    Console.ReadLine();
}

public static void ProcessPhaze1()
{
    Thread.Sleep(1000);

    barrier.SignalAndWait(2000);
}
```

Here each thread calls `ProcessPhaze1` from `Main`. Thus `ProcessPhaze1` will be called in parallel by five threads. Once each of them finishes execution (for simplicity we have just used `Thread.Sleep`) it signals Barrier to wait and thus it stops itself. As shown here, you can specify the timeout value for `SignalAndWait` to ensure that deadlock does not occur.

Once all threads are signalled, it calls the call back specified as argument in Barrier. The first argument 5 specifies how many threads are participant of the Barrier. Here we are using five threads. If we change the argument to 2, it will print the `Phaze2` call back twice. The argument specifies how many times callback needs to be called for each signal received. After each call back is executed, Barrier `PhazeNumber` gets incremented. You can get the value of the Phaze number from `barrier.CurrentPhaseNumber`.

Barrier also supports dynamic addition and removal of participants.

What is SynchronizationContext?

In the Windows environment, **Thread Affinity** is one of the important concerns that every developer comes across. The UI elements on Windows applications are affinated towards the Thread that creates it. It ensures that you always access the UI element from the Thread on which it is created. The UI is generally created on UI thread, and hence every UI element needs to be accessed on the UI thread. Windows Forms applications have been using `ISynchronizeInvoke` interface to invoke the code to run on the thread in which it has been created. .NET introduces a new model for thread synchronization using `SynchronizationContext`. The new Threading model simplifies the communication between other threading models and simplifies the synchronous and asynchronous operations.

`SynchronizationContext` is a new class recently introduced and is widely used inside framework class libraries to communicate between different threading models. The new **async** language implementation already uses `SynchronizationContext` inside its core to identify correct thread to post messages. `SynchronizationContext` represents the abstract base class for a model which defines two methods:

- ▶ `Send`: This invokes a delegate which is passed to it synchronously
- ▶ `Post`: This invokes a delegate passed as an argument asynchronously

Each of them also takes a delegate of type `SendOrPostCallback` and the state object. When `SynchronizationContext` is created, it associates the current thread to its context and keeps the reference inside the object, so that, based on the thread in which the block is executing, the `SynchronizationContext.Current` will always get you the object that is created in the current thread.



The `Post` actually uses `ThreadPool` internally to call the call back you pass asynchronously.

Now let's create a class that uses `SynchronizationContext`:

```
public class MyCallerType
{
    private Thread currentThread;

    private SynchronizationContext context;

    public event EventHandler EventCall back;

    public MyCallerType()
    {
        context = SynchronizationContext.Current;
        context = context ?? new SynchronizationContext();

        currentThread = new Thread(new Threadstart(Run));
        currentThread.Start();
    }
}
```

```

    }

    private void CallEventHandler(object state)
    {
        EventHandler handler = EventCall back;

        if (handler != null)
        {
            handler(this, EventArgs.Empty);
        }
    }

    private void Run()
    {
        context.Send(new SendOrPostCall back(this.CallEventHandler),
null);
    }
    private void RunAsync()
    {
        context.Post(new SendOrPostCall back(this.CallEventHandler),
null);
    }
}

```

The class `MyCallerType` actually holds the current `SynchronizationContext` type. So when `Run` or `RunAsync` is called, it gets the current thread from its `SynchronizationContext` (which is held from its object creation) and invokes it.

See also

- ▶ <http://bit.ly/ThreadingTutorial>
- ▶ <http://bit.ly/SynchronizationContext>

Working with Event-based asynchronous pattern and `BackgroundWorker`

EAP is a model that has also been introduced to handle threading in an easier and elegant way. The Event-based asynchronous pattern forms few rules that you need to follow while following the pattern. The implementation of Event-based Asynchronous pattern has been widely accepted, which uses events to notify the caller with the changes to the thread.

Getting ready

In this recipe, we are going to show an example of the `BackgroundWorker` type rather than implementing a new EAP class which will use `ThreadPool` in the background and run the method that is passed to it asynchronously. The `BackgroundWorker` class has special features like `ProgressReport`, `CompleteCallback`, or even cancellation of the call, hence it becomes the sole EAP type which most of the developers use to take benefit of.

How to do it...

Let us create a Windows application to show the usage of the `BackgroundWorker` class in the recipe.

1. Create a method named `XXXAsync` for asynchronous member with the same parameters that are needed to be called for the synchronous member. You can pass a `State` object as well if needed. This method calls `BeginInvoke` to perform the asynchronous operation.
2. `AsyncCompletedEventArgs`, used as the second parameter, is inherited from `EventArgs` and represents the additional settings that need to be passed to every event when it completes. The `AsyncCompletedEventArgs` parameter itself exposes properties like `Cancelled` (which indicates whether the operation is cancelled), `Error` (holds the error object) or `UserState` (holds the state of the call).
3. Create a delegate with `XXXCompletedEventHandler`, which takes argument as an object, and `XXXCompletedEventArgs`.
4. Create an event of type `XXXCompletedEventHandler`. The event is raised when the asynchronous operation is complete.
5. `BackgroundWorker` is a managed class that has already implemented the EAP. We use `BackgroundWorker` in the following code to implement the asynchrony in the application:

```
BackgroundWorker backgroundWorker;  
BackgroundWorker Worker  
{  
    get  
    {  
        this.backgroundWorker = this.backgroundWorker ?? new  
BackgroundWorker();  
        return this.backgroundWorker;  
    }  
}  
public long DoCpuIntensiveWork()  
{  
    int i = 2, j, rem, result = 0;
```

```
while (i <= 1000000)
{
    for (j = 2; j < i; j++)
    {
        rem = i % j;
        if (rem == 0)
            break;
    }
    if (i == j)
        result = i;
    i++;

    int progress = (int)((float)i / (float)1000000 * 10000);
    if (progress > 0)
        this.Worker.ReportProgress(progress);

    if (this.Worker.CancellationPending)
        return i;
}
return result;
}

private void start_Click(object sender, EventArgs e)
{
    this.Worker.RunWorkerAsync();
    this.button1.Enabled = false;
    this.button2.Enabled = true;
}

public void backgroundWorker_DoWork(object sender, EventArgs e)
{
    this.DoCpuIntensiveWork();
}

private void Form1_Load(object sender, EventArgs e)
{
    this.Worker.WorkerSupportsCancellation = true;
    this.Worker.WorkerReportsProgress = true;
    this.Worker.DoWork += new DoWorkEventHandler(backgroundWorker_DoWork);
    this.Worker.ProgressChanged += new ProgressChangedEventHandler(Worker_ProgressChanged);
    this.Worker.RunWorkerCompleted += new RunWorkerCompletedEventHandler(Worker_RunWorkerCompleted);
}
```

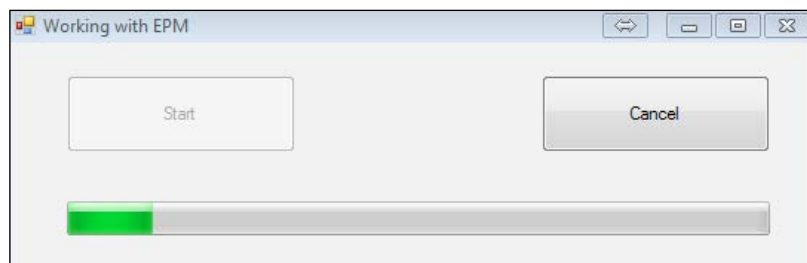
```
}

void Worker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    this.button1.Enabled = true;
    this.button2.Enabled = false;
}

void Worker_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
    this.progressBar1.Value = e.ProgressPercentage;
}

private void cancel_Click(object sender, EventArgs e)
{
    this.Worker.CancelAsync();
}
```

The previous code uses `BackgroundWorker` to implement the functionality of Asynchronous operation. We call `DoCPUIntensiveOperation` from the `DoWork` eventhandler of `BackgroundWorker`. `DoWork` represents the working module. The `ProgressChanged` event is raised when we call the `ReportProgress` method from inside the working module where we need to update the progress bar control on the UI. You should also note that the event `ProgressChanged` automatically detects the right thread from where it is created.



This form displays two buttons which let us start a long running process with the click of the **Start** button and the operation progress is reflected on the progress bar, and the **Cancel** button cancels the operation.

- When **Cancel** is clicked we request cancellation. The logic inside `DoWork` should check for `CancellationPending`, and depending on its value, it cancels the operation.
- `RunWorkerCompleted` is called when the operation has finished execution.

How it works...

Event-based Asynchronous pattern works similar to any other asynchronous pattern, but uses events and delegates to implement Call backs. The `BackgroundWorker` class being the classic example of EPM, implements the Event-based Asynchronous pattern. According to this pattern, the class should contain:

- ▶ A method with `MethodNameAsync` to invoke the operation.
- ▶ There should be a corresponding `MethodNameCompleted` event that will be invoked when method has finished execution. This serves as a call back to the method.

By the way, you should also remember that the `RunWorkerAsync` immediately returns the control, as it runs the method asynchronously. It creates a new `Thread` from the `ThreadPool` and calls it as a `WorkItem` to the `ThreadPool` thread. It also subscribes a call back that will invoke the `RunWorkerCompleted` event when the method has finished execution.

Additionally, the `BackgroundWorker` class can also report progress of the execution if you call `ReportProgress` from your code, which will invoke the `ProgressChanged` event as we saw before. There are lots of other features of the `BackgroundWorker` class too, and it is recommended to have Event-based Asynchronous Pattern type following .NET threading patterns.

See also

- ▶ <http://bit.ly/BackgroundWorker>
- ▶ <http://bit.ly/BWUseCases>

Working with thread locking and synchronization

Locking is an essential part of a multi-threaded application. Synchronization on the other hand is another important factor of a program that ensures the smooth running of the program in a managed environment. In this recipe, we are going to cover how to use thread locking and synchronization feature of .NET to avail the smooth running of the program logic.

While running a multi-threaded application which sometimes involves multi-core systems, resource sharing is one of the important concerns for developers. While being in concurrent access, mutual exclusion exists with the use of lock block of C#. In this recipe, we are going to cover how to employ locking on a portion of code that requests mutual exclusiveness of the logic even when the application supports parallelism.

Getting ready

Thread synchronization is an important technique that enables coordination of two or more threads with a predictable outcome. Generally when we start an asynchronous operation there is no rule of thumb that defines the outcome. Processor allocates dedicated threads based on the scheduling and processor architecture. Running an innumerable number of threads might cause a problem. Ideally, each core should have only one thread. Threads running in parallel need blocking to perform synchronization and coordination between multiple threads. Let us create a multi-threaded application and try out the features.

Synchronization is of the following four types:

- ▶ **Simple blocking methods:** These types of blocking methods wait for another thread to finish or wait for an amount of time. For instance, `Thread.Sleep`, `Thread.Join`, `Task.Wait`, and so on.
- ▶ **Locking constructs:** These constructs limits the number of threads that execute a block of code at a certain time. Exclusive locking constructs allows only one thread to run a specific code at a time. This lets other threads access common data (some shared objects) without interfering with each other. For instance, `lock`, `Mutex`, `SpinLock`, and so on. Some of the non-exclusive locking constructs are `Semaphore`, `Reader/Writer` locks, and so on.
- ▶ **Signaling construct:** These allows you to pause until one thread receives some notification. This has already been discussed earlier.
- ▶ **Non-blocking synchronization constructs:** These protect access to a common field by calling process or primitives. For instance, `Thread.MemoryBarrier`, `Thread.VolatileRead`, the `Interlocked` class, and so on.

How to do it...

In this recipe, let us look into locking constructs and synchronization feature of .NET parallel world.

1. Threads trying to update a shared resource (say a static variable) often gets bad data when accessing it concurrently. Lock statement makes some section of the code exclusive for a thread to execute.

```
class Threadsafe
{
    static readonly object locker = new object();
    static int val1, val2;

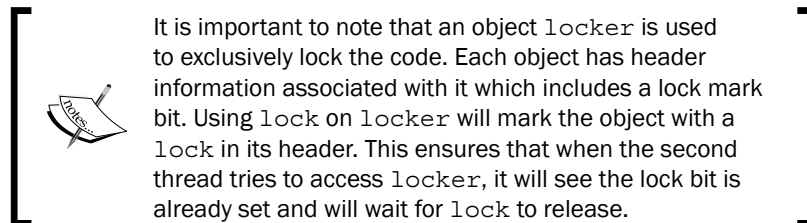
    static void Go()
    {
        lock (locker)
```

```

    {
        if (val2 != 0) Console.WriteLine (val1 / val2);
        val2 = 0;
    }
}

```

The previous code exclusively locks using an object named `locker`. The `lock` construct will ensure that the calculation on static integer variable `val1` and `val2` will be exclusively locked and cannot be accessed by more than one object.



2. Semaphores work a little different to locks. In the case of `Semaphore`, any thread can release a lock, but it has a restriction to allow a certain number of threads to access it concurrently based on the capacity. Let us look into the following code:

```

static SemaphoreSlim sem = new SemaphoreSlim (3);
static void SemaphoreTest ()
{
    sem.Wait();
    // the code that can only run by 3 threads at a time
    sem.Release();
}

```

Here if we call `semaphoreTest` more than three times, the rest of the threads will be queued, as `_sem.Wait` allows only three threads to run the code between it and release at a time.

3. To implement automatic locking of a `Threadsafe` object, we need to derive the class from `ContextBoundObject` and apply the `Synchronization` attribute to the class.

```

[Synchronization]
public class AutomaticThreadsafe : ContextBoundObject
{
    public void Demo()
    {
        //Locks exclusively
    }
}

```


The class in this code is automatically set to `Threadsafe`. Execution of any code within an object that is derived from `ContextBoundObject` and application of the `Synchronization` attribute will ensure the object to be `Threadsafe` automatically.

4. The code in step 3 automatically locks the execution of code within its methods to ensure only one thread can exclusively access the object.
5. The `Threadsafe` object is sometimes called `reentrant`. If you pass `True` as a `reentrant` argument to the `Synchronization` attribute, the thread will release the locks temporarily when execution leaves the context.
6. The `reentrant` threads prevents deadlocks, but allows more than one thread to call a method when the context switches.

How it works...

`Lock` is an implementation of `Monitor.Enter` and `Monitor.Exit`. If you see the `Threadsafe` code actually, the equivalent code will look like the following:

```
Monitor.Enter (locker);
try
{
    if (val2 != 0) Console.WriteLine (val1 / val2);
    val2 = 0;
}
finally { Monitor.Exit (locker); }
```

This code is exactly same as the `lock` statement, the runtime evaluates the same code in both the cases.

`Monitor` has a method called `TryEnter`, which has a `Timeout` seed overload that can be used to try locking an object and wait for a timeout.

The automatic synchronization of an object is done using `ContextBoundObject`. The object that has its parent as `ContextBoundObject` is intercepted to all its properties and methods. The CLR automatically creates a proxy of the object which acts as an intermediary.

There's more...

Now let us consider looking into some of the other things that you need to know about locking and synchronization. Let us discuss them.

How to determine Mutex of a Thread?

Mutex is not new to the .NET framework. The Win32 API supports an object called Mutex and .NET framework class `Mutex` is actually a wrapper class for the same Win32 kernel `Mutex` object. `Mutex` objects are generally used to synchronize threads across processes. `Mutex` objects can be named and hence you can signal whether the `Mutex` is acquired or not by some other thread (probably in a different process). You should note that `Mutex` is created with the call of the Win32 `CreateMutex` call. You can check ownership of `Mutex` using `WaitAll`, `WaitAny`, and so on calls and which avoids deadlocks.

As we have already discussed, `Mutex` has an underlying Win32 kernel object associated with it, even though it supports cross-process thread synchronization, yet it is heavier than the `Monitor.Enter/Monitor.Exit` sequence, and you need to explicitly close `Mutex` when it is released.

Let us use `Mutex` in code to clear how you could use `Mutex`.

```
static Mutex _mlock = new Mutex();
public static long Sum()
{
    long counter = 0;
    if (_mlock.WaitOne(1000))
    {
        try
        {
            for (int i = 0; i < Repository.Count; i++)
                Interlocked.Add(ref counter, Repository[i]);
        }
        finally
        {
            _mlock.ReleaseMutex();
        }
    }
    return counter;
}

public static void AddToRepository(int nos)
{
    if (_mlock.WaitOne(2000))
    {
        try
        {
            for (int i = 0; i < nos; i++)
                Repository.Add(i);
        }
        catch { }
        finally
    }
```

```
        {  
            _mlock.ReleaseMutex();  
        }  
    }  
}
```

In this code we have created `Mutex`, and `lock` is acquired using the `WaitOne` method of the `Mutex` object. We have also specified the timeout value for `Mutex`. If the timeout is elapsed, `Mutex` returns `False`.

If you have more than one `Mutex` that needs to be acquired before running a code, you can put all the `Mutex` objects in an array and call `WaitHandle.WaitAll` to acquire all the `Mutex` objects at a time.

```
Mutex[] mutexes = { this._mlock, this._mlock2, this._mlock3};  
if (WaitHandle.WaitAll(mutexes))  
{  
    // All mutexes acquired.  
}
```

The `WaitHandle` method actually avoids the deadlock internally.

Similar to `WaitAll`, the `WaitHandle` class also supports methods, such as the `WaitAny` or `SignalAndWait` methods to acquire locks.

Use Mutex for Instance Count

As I have already told you, `Mutex` objects can be named and can be accessed across process, you can use `Mutex` to count the number of instances of a certain thread that is created on a particular machine. Let's see how to do this:

```
bool instanceCountOne = false;  
Mutex mutex = new Mutex(true, "MyMutex", out instanceCountOne);  
using (mutex)  
{  
    if (instanceCountOne)  
    {  
        // When there is only one Instance  
    }  
}
```

Here we name the `Mutex` as `MyMutex` which will count all the existing processes of the system, and see whether the `MyMutex` object is newly created or it's already there. `instanceCountOne` identifies whether the name is newly created or not. Thus, it will identify the instance count of `Mutex`.

If you write this code at the entry point of your application, you can check out how many instances of the application actually exist in the system.

Locking using Spinlock

SpinLock like the `Monitor.Enter/Monitor.Exit` sequence spins a number of CPU cycles when it tries to acquire a lock. `Monitor.Enter/Monitor.Exit` on the other hand uses some CPU cycle initially, but also invokes a True wait after a certain timeout if lock cannot be acquired. In case of multi-core machines, the Wait timeout for Monitor is actually very short and in this scenario it is better to use SpinLock instead of Monitor. But it is recommended to check the CPU performance before using SpinLock in profiler, as it is natural that Spinlock always consumes CPU memory before acquiring the lock.

```
static SpinLock _spinlock = new SpinLock();
public static List<int> Repository = new List<int>();
public static long Sum()
{
    bool lockTaken = false;
    _spinlock.Enter(ref lockTaken);
    long counter = 0;
    try
    {
        for (int i = 0; i < Repository.Count; i++)
            Interlocked.Add(ref counter, Repository[i]);
    }
    finally
    {
        if (lockTaken)
            _spinlock.Exit();
    }
    return counter;
}
public static void AddToRepository(int nos)
{
    bool lockTaken = false;
    if (_spinlock.IsHeld)
        return;
    _spinlock.TryEnter(1000, ref lockTaken);
    try
    {
        for (int i = 0; i < nos; i++)
            Repository.Add(i);
    }
    catch { }
    if (lockTaken)
        _spinlock.Exit();
}
```

`SpinLock` has an `Enter` and `Exit` to acquire and release the lock. It requires a `Boolean` field to be passed to it which ensures whether the lock is actually acquired or not. `TryEnter` is used to specify the timeout seed for the lock.

As I have already told you, `SpinLock` will yield a few time slices when it hasn't acquired the lock, this will ensure that the garbage collector makes progression on the block that tries to acquire the lock. Another important point is that `SpinLock` is a structure, and if more than one thread requires to check the lock, it should be passed by reference.

`SpinLock` also allows you to check whether the lock is held by any thread using the `IsHeld` property or even you can check whether the current thread holds the lock using the `IsHeldByCurrentThread` property. If more than one thread tries to acquire the locked resource, `SpinLock` generates `LockRecursionException`, and hence `SpinLock` is not reentrant.

ReaderWriterLock for non-blocking synchronization constructs

`ReaderWriterLock` allows shared locks together with exclusive locks. Hence it is possible to read the same resource using shared locking, each time `ReaderLock` is invoked the lock count is increased, but allows you to read the block of code inside it. `ReaderWriterLock` also gives you an option for exclusive locking using the `WriteLock` statement to eliminate inconsistent reads. Let's see how to use this with code.

```
static ReaderWriterLock locker = new ReaderWriterLock();
public static List Repository = new List();

public static long Sum()
{
    locker.AcquireReaderLock(1000);

    long counter = 0;
    try
    {
        for (int i = 0; i < Repository.Count; i++)
            Interlocked.Add(ref counter, Repository[i]);
    }
    finally
    {
        locker.ReleaseReaderLock();
    }
    return counter;
}

public static void AddToRepository(int nos)
{

```

```

        if (locker.IsWriterLockHeld)
            return;

        locker.AcquireWriterLock(1000);

        try
        {
            for (int i = 0; i < nos; i++)
                Repository.Add(i);
        }
        catch { }

        locker.ReleaseWriterLock();
    }

```

AddToRepository is only used to add integer values to a list. You can easily lock the repository using the `lock (Repository)` statement, and that will mean an exclusive locking will be established. On the other hand, the code written inside AddToRepository also produces exclusive locking using the `AcquireWriterLock` statement. You can optionally specify the timeout value for the `WriterLock` (specify `TimeSpan.Infinite` when you don't need timeout).

Putting an exclusive lock on the list during the sum operation can produce inefficiencies on code and should be avoided. We have used `AcquireReaderLock` in this case to create shared locking for the block. Hence, concurrent reads can take place only when no `WriterLock` is established, but concurrent writer locks are not possible.

`ReaderWriterLock` also has a method to escalate from Reader to Writer and vice versa using `UpgradeToWriterLock` or `DowngradeFromWriterLock`.

Every lock should always specify its `Release` statement finally.

See also

- <http://bit.ly/ThreadLock>

Lock statement using task-based parallelism in concurrent programming

Concurrent programming, as we have seen in common threading patterns, is being simplified day-by-day. In .NET 4.0 Microsoft introduced a new complex type `Task` that simplifies the asynchrony more than ever before. The concurrency has been separated with asynchrony and the task is named as something that is expected in future. Every asynchronous operation is specified with a task either by using CPU cycles, an input/output operation, a Network card, or even some other device. The tasks are identified by `SynchronizationContext` when the operation has finished its execution and gets you the result to the calling environment.

The **Task Parallel Library (TPL)** inherently uses `ThreadPool` behind the scenes, but is supported with algorithms that adjust the number of threads that should be running on the system automatically to maximize throughput. The tasks are lightweight and more scalable to system resources, and hence are the preferred choice for both concurrent applications that involve CPU processing, and other asynchronous operations. TPL also enhances the programmatic control with the thread or work items by supporting features such as APIs for task waiting, cancellation and continuations, robust exception handling, detailed status notifications, custom scheduling, and so on.

In this recipe we are going to cover the basic usage of the Task Parallel Library and see how to use it in your daily asynchronous programming needs.

Getting ready

To start with this recipe, let us create a Console Application. Let us start by writing a small piece of code:

```
Parallel.Invoke(FirstMethod, SecondMethod);
```

This code actually invokes the two methods, `FirstMethod` and `SecondMethod` in parallel using `ThreadPool`. So if you put a `Thread.Sleep` inside the methods, it will produce a delay when calling the method. `Parallel.Invoke` takes an array of action delegates, which will be called in parallel. `Parallel.Invoke` also takes a second argument of `ParallelOptions` which allows you to specify the degree of parallelism, cancellation options, and so on. We will look into it in more detail in this recipe.

How to do it...

1. A Task is a combination of `ThreadPool` with some algorithms that support easier constructs and abstraction to work with multi-core systems and parallel computing. The basic construct of creating a task is as follows:

```
Task t1 = new Task(FirstMethod);  
t1.Start();
```

This code creates an object of task which will call `FirstMethod`. The call is made when `Start` is called for the task object just like how a thread works.

2. A factory is provided with the TPL that can also be used directly to create a new task. You can also use `Task.Factory.StartNew(SecondMethod)` to call `SecondMethod` similar to the previous operation. In the second case the task object will be created automatically and the `Start` method will be called immediately.

3. If we need to get the output of a method, we need to check the `Result` property. `Task` has a few more constructors that allow you to pass the generic type parameters that determine the return object of the method. This ensures that the parameters are type safe and yet does not involve unnecessary reference conversion.

```
Task<int> tThird = new Task<int>(ThirdMethod);
tThird.Start();
```

```
if (tThird.Status == TaskStatus.RanToCompletion)
    Console.WriteLine(tThird.Result);
```

In this code `Func<int>` has been passed rather than the action. `ThirdMethod` actually returns an integer value which we can get from the `Result` property of the task object. The status indicates the various states of the task object, among which, a few are: `Created`, `Running`, `Cancelled`, `Faulted`, `RanToCompletion`, and so on. In fact if you look carefully, the call to `Result` actually blocks the execution until the method returns. Thus, even if the status is not evaluated to `RanToCompletion` it can still call `Result` without error.

4. `Task` supports continuation. The `ContinueWith` method of the task lets you specify the call back when the execution of a task completes. Each `ContinueWith` method actually receives the object of antecedent task object and hence the entire chain is maintained. You can get the result of a task from its previous chain and call the next method on the chain.

```
Task<int> finalValue = Task.Factory.StartNew(FirstMethod)
    .ContinueWith((x) = SecondMethod())
    .ContinueWith((y) = ThirdMethod());
Console.WriteLine(finalValue.Result);
```

In this code, `FirstMethod` is called which calls `SecondMethod` as its call back and finally `ThirdMethod` to get the final result. Notice, the `ContinueWith` method takes an `Action<Task>` to receive the previous task object to continue execution of the call back. Hence, the whole chain is maintained.

5. `Task` supports waiting. As I have already told you, the call of `Result` from a task object actually waits for the result, you can also call the `Wait` method of the task object to actually wait for the completion of the task. When dealing with multiple tasks you can use `Task.WaitAll` to wait for all tasks to complete.

```
Task t1 = new Task(FirstMethod);
Task t2 = new Task(SecondMethod);
Task<int> t3 = new Task<int>(ThirdMethod);

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

Task.WaitAll(t1, t2, t3);
```


In this code, three task objects have been created and `Task.WaitAll` is called. This will ensure that it waits for all tasks to finish. You can also call `WaitAny` which exclusively waits for only one method.

6. Tasks also support `CancellationToken` to cancel a task when some external cancellation has been set. Remember, cancellation of a task is not something equivalent to `Thread.Abort`. Task allows you to pass a token which needs to be manually handled and ensures that the wait is released only when the program is stable.

```
public static void CancellationOperation()
{
    CancellationTokenSource source = new
CancellationTokenSource();
    CancellationToken token = source.Token;

    Task t1 = new Task(() => DoWork(token), token,
TaskCreationOptions.LongRunning);
    t1.Start();

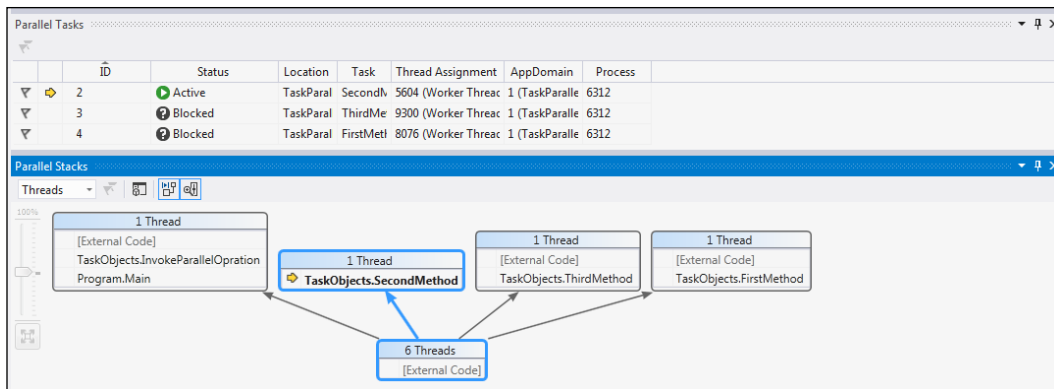
    Thread.Sleep(10000);
    source.Cancel();
}

static void DoWork(CancellationToken token)
{
    while (true)
    {
        Console.WriteLine("Current Time is {0}", DateTime.Now);
        Thread.Sleep(1000);
        if (token.IsCancellationRequested)
        {
            Console.WriteLine("Cancellation has been requested");
            break;
        }
    }

    Console.ReadKey(false);
}
```

This code passes a cancellation token to the `DoWork` method. The `DoWork` method continually checks whether the cancellation has been requested or not and depending on its value it breaks the never ending loop. `CancellationToken` is actually created using `CancellationTokenSource`, which can invoke `Cancel` to cancel the operation. In the previous code, the cancellation is invoked after 10 seconds.

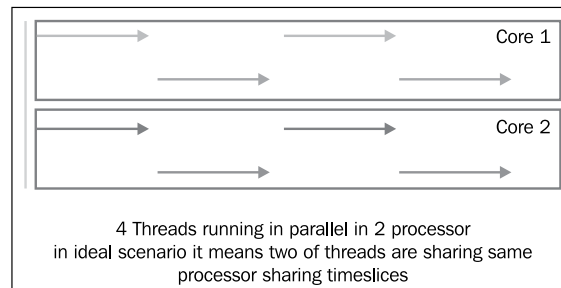
7. Every task has a few options, which you can pass. For instance, in the code in step 6 the `LongRunning` task will indicate that this task is long running and any short running task will be invoked in between. `PreferFairness` will make tasks that started before to finish before, and `AttachToParent` makes the task attached with the current task.
8. Each task is associated with an ID which identifies the task in Visual Studio debugger. As we have already shown before, the **Parallel Stacks** and **Parallel Tasks** windows help to identify these Tasks uniquely and view its information.



The **Parallel Tasks** window shows all the tasks that are running in parallel and the **Parallel Stacks** window shows the graphical representation of the tasks running on different threads.

How it works...

Task Parallel Library internally runs on ordinary `ThreadPool` or threads. It uses enhanced algorithm to actually handle parallel execution easier and elegantly. The performance of a code running TPL is often better than a code that does not use it. We live in a world in which most CPUs have at least two cores. Scheduling threads into multiple CPUs often becomes a difficult job for a programmer when you are continuously dealing with throughput. TPL supports multiple CPU execution as well and automatically manages this for a program.



In the preceding figure, it is depicted how threads are generally scheduled by the Task Parallel Library. If you are running four threads in parallel in an ideal situation and if there are two processors installed, the two threads will run in parallel, but the two of them share time slices of one single processor.

TPL supports Data Parallelism or Task Parallelisms. You can run your logic in parallel around your data or while processing your data using the `Parallel` class members:

```
Parallel.For(0, 100, ProcessMe);
```

The preceding code loops through 0 to 100 to call the method `ProcessMe` with the current value of index as argument in parallel. Hence, if you print the index inside the `ProcessMe` method, you will see the method runs in parallel. Similar to this, you can replace the `ForEach` loop with `Parallel.ForEach` or methods with `ThreadLocal` data.

Task Parallelism actually works on task and its associated objects. It works as an operation which is similar to threading and acts as a new threading pattern. Task objects are used widely with **async** to implement asynchrony better.

There's more...

Task parallelism library has made certain common use cases very easy to implement. As we move ahead in using it, more and more use cases come to mind. Let us discuss some of the additional information about the library which is nice to have for anyone.

How to handle exceptions inside a Task?

Unlike most of the existing threading patterns, Task Parallel Library has enhanced support of exception handling within it. Every task exposes one `Exception` object and `Status` such that when status is `Faulted`, or the operation throws an exception, the `Task` automatically receives it and stores it into the `Exception` property.

```
Task<int> t1 = new Task<int>(ThirdMethod);
t1.Start();
t1.Wait();
if (t1.Status == TaskStatus.Faulted)
{
    foreach(var exception in t1.Exception.InnerExceptions)
        Console.WriteLine(exception.Message);
}
```

`Task` exposes the `Exception` object which actually receives an `AggregateException` object. This exception is especially created for TPL and is used to store all its inner exceptions into the `InnerExceptions` object. Here in the preceding code, the task throws an exception and all the exceptions that have been thrown either by the `Task` or its child tasks will be shown on the console.

How to work with TPL without concurrency?

TPL supports the call of `BeginXXX/EndXXX` APIs too internally which do not create a new thread automatically inside, but rather, use the existing `Begin` and `End` patterns to work on other device operations. For instance, if you are working on `Stream`, `Socket`, or even `WebRequest`, `Task.Factory.FromAsync` allows you to call these operations without creating new threads and blocking them. The async operations in such cases will work synchronously, but will use `SynchronizationContext` to find underneath device call backs (for example I/O Completion Ports on Windows).

```
NetworkStream stream;
byte[] data;
int bytesRead;
Task<int> readChunk = Task<int>.Factory.FromAsync (
    stream.BeginRead, stream.EndRead,
    data, bytesRead, data.Length - bytesRead, null);
```

Here `FromAsync` will read data from the network without blocking a separate thread and everything works synchronously.

More easier construct of these is available with `async` and `await` patterns discussed in the next recipe.

How to create custom TaskScheduler?

`Task Parallel Library` itself has created a `TaskScheduler` that schedules `Tasks` into multiple threads and also into multiple cores, and also looks after the overall performance of the program to maximize throughput. You can access the default `TaskScheduler` using `TaskScheduler.Default`. The current execution context `TaskScheduler` is available from `TaskScheduler.Current` or using the `FromCurrentSynchronizationContext` method. By the way, if you are working with advanced scenarios, you might also create your own `TaskScheduler` and schedule the tasks based on your own requirement.

```
public class ThreadPerTaskScheduler : TaskScheduler
{
    protected override IEnumerable<Task> GetScheduledTasks() { return
        Enumerable.Empty<Task>(); }

    protected override void QueueTask(Task task)
    {
        //Execute each Task into individual Thread
        Thread th = new Thread(() = TryExecuteTask(task));
        th.IsBackground = true;
        th.Start();
    }
}
```

```
protected override bool TryExecuteTaskInline(Task task, bool
taskWasPreviouslyQueued)
{
    return TryExecuteTask(task);
}
}
```

In the preceding code, `ThreadPoolScheduler` actually creates one thread per task. That means it creates each individual orphan thread (not even from `ThreadPool`) and starts the task on it. You should notice that I have implemented the `TaskScheduler` by deriving from the `TaskScheduler` class, and `QueueTask` is called when the task should be executed.

To use a custom `TaskScheduler` class, you need to pass the object when you create the object of task. If you do not specify a `TaskScheduler` object, the .NET framework uses the default one.

How to deal with PLINQ?

PLINQ or parallel LINQ is an integration of parallel into LINQ queries. When working with PLINQ, we convert the `IEnumerable` into `ParallelEnumerable`. An extension method that is attached with the `Enumerable` objects, called `AsParallel`, allows you to get `ParallelEnumerable` from an `IEnumerable` object.

```
var sequence = Enumerable.Range(1, 10000);

var evens = from n in sequence.AsParallel()
            where (n % 2) > 0
            select n;
```

In the preceding code the LINQ query uses `AsParallel` to get `ParallelEnumerable` which runs in parallel, and each number is divided to get the sequence of even numbers.

`ParallelEnumerable` has few extension methods which are important to note. You can use `AsOrdered` to preserve ordering while executing the parallel LINQ query, or you can specify how many processors the LINQ will use using `WithDegreeOfParallelism`.

```
var sequence = Enumerable.Range(1, 10000);

var evens = from n in sequence.AsParallel().AsOrdered().
            WithDegreeOfParallelism(2)
            where (n % 2) > 0
            select n;
```

The preceding code will preserve the ordering of the `Enumerable` objects and will parallelize the query into two physical processors. The degree of parallelism is often important while tuning the query performance, as parallelize does not always mean an increase in performance.

See also

- ▶ <http://bit.ly/TPLLibrary>

Working with async and await patterns

.NET asynchronous APIs have always been an element of problem to developers. Understanding a code that has been written in synchronous cannot be easily transformed into asynchronous. There are a lot of things that need to be done before making the actual code work. If you are following some pattern in code, making it asynchronous makes it inside out. You need to deal with a lot of call backs and methods such that it behaves the same way as it did earlier. This way the maintenance of code becomes a tough task for anyone.

.NET places one step forward to simplify this long running problem that has been a continuous need of technology. It introduces some compiler fixes which will keep things simple for the programmer, but holding the underlying asynchrony to work in the same way as it did with synchronous. If you have an existing code that runs in a synchronous way, making it asynchronous is just a little fix. You don't need to change any logic of the code after introducing the new asynchronous pattern. In this recipe I am going to introduce this new pattern with the help of simple examples.

As C# is continually growing with branches, the latest release of C# 5.0 introduces the new async feature considering one of the major problems that programmers often face while developing asynchronous code. The main motive though is to mould the language in such a way that the asynchronous kind of programming pattern does not need to do any additional coding or follow some specific style while converting it from the synchronous form. During the inception of C# 2.0, the async pattern was introduced using `BeginInvoke` and `EndInvoke` calling patterns of code. But if you look closely, the whole logic of code needs to be modified and rewritten to support the new pattern when you are going to change your code from a synchronous one. You need to consider creating the call back method and change the logical flow of the program completely. C# 5.0 keeps the notion of logical program flow intact, but introduces two keywords to transform the same code instantly into asynchronous form. Asynchrony is completely different than concurrency. Concurrency is a subset of asynchrony where the operation on which the application is going to run asynchronously is CPU bound. The async pattern uses `SynchronizationContext` to find an appropriate message queue automatically to bridge the gap between the unmanaged environment and the managed environment.

Getting ready

Every asynchronous operation using the new asynchronous pattern is ornamented with an `async` modifier on the function and those calls can be ornamented using the `await` keyword in code. The `async` will make the method *awaitable*, that means while calling the method, you can use the contextual `await` keyword before that, such that the control waits for the operation to finish asynchronously. The method that has `async` pattern implemented can support only three types of return statement. You can return `void` or a **Task** with the support of generic task object in form of `Task<T>`. Let us take an example of such an operation:

```
public async Task WaitAsync()
{
    await Task.Delay(1000);
    Console.WriteLine("After 1 Second");
    await Task.Delay(1000);
    Console.WriteLine("After 2 Seconds");
}
```

In the preceding code the `async` method has been implemented. The method simply puts a `Console` statement initially after one second and then after another second. The code looks like same as `Thread.Sleep(1000)`, but actually it isn't. In this code, the **Task** object is returned as soon as the first `await` is encountered. The thread automatically yields the **Task** object on which you can either `await` again from outside, or create a call back to it. It does not block the current thread for 1 second, rather it immediately returns the **Task** object and makes the current thread available to the programmer. The new asynchronous pattern does not employ creation of a new thread as well, rather it creates an object of `SynchronizationContext` and puts a call back using its `post` method such that when the `Delay` has finished, it posts from the context automatically and invokes the next call on the same thread.

To start using `async` we first create a **Windows Presentation Foundation (WPF)** application. We use a WPF application to get data from the Internet asynchronously and use this to model our concept to build a more sophisticated application.

In this application we are going to create a few buttons which download data from a URL using the Internet. To work with the web, we need `WebClient`. `WebClient` is a class that exists in the `System.Net` namespace which lets us work with web from client space. Let us put this further by creating the application for the recipe.

How to do it...

Let us now create an application to demonstrate this new feature.

1. Start a WPF application and create a UI. In MainWindow.xaml we put few buttons inside StackPanel.

```
<StackPanel Orientation="Vertical">
    <Button x:Name="btnSync" Content="Synchronous"
Click="btnSync_Click" />
    <Button x:Name="btnaSyncPrev" Content="Traditional Async"
Click="btnaSyncPrev_Click" />
    <Button x:Name="btnaSyncPres" Content="Mordern Async"
Click="btnaSyncPres_Click" />
    <Button x:Name="btnaSyncPresParallel" Content="Mordern
Async Parallel" Click="btnaSyncPresParallel_Click" />
    <Button x:Name="btnGoCPUBound" Content="CPU Bound
Call(Uses ThreadPool Thread)" Click="btnGoCPUBound_Click" />
    <ProgressBar Maximum="100" x:Name="prg" MinHeight="30" />
    <TextBlock x:Name="tbStatus" />
</StackPanel>
```

StackPanel creates a stack of buttons, Textblock to show the status, and ProgressBar to show progress.

2. Each of the buttons are assigned to their own EventHandler which will be called when they are clicked. For all the buttons we do the same thing, that is, we call a server URL 10 times and create a string from the response. Using the Synchronous button we download the string one by one synchronously.

```
public void SynchronousCallServer()
{
    WebClient client = new WebClient();
    StringBuilder builder = new StringBuilder();
    for (int i = 2; i <= 10; i++)
    {
        this.tbStatus.Text = string.Format("Calling Server
[{0}]..... ", i);
        string currentCall = string.Format("Feed, {0}", i);
        string rss = client.DownloadString(new Uri(currentCall));

        builder.Append(rss);
    }
    MessageBox.Show(string.Format("Downloaded Successfully!!!
Total Size : {0} chars.", builder.Length));
}
```


Obviously the preceding code hangs up the whole application when the synchronous call is made. As an end user, you don't want to see such an application and the code is almost unusable in a production environment.

3. The second button does the same thing, but by calling the `Async` method directly. In the traditional asynchronous pattern, a new thread is created from `ThreadPool` and is blocked waiting for `WebClient` to finish its operation.
4. In the third button, the new async pattern is used. In case of modern approach the call is made asynchronously from the UI thread which immediately returns the task object after calling the network.

```
public async Task AsynchronousCallServerMordernAsync()
{
    WebClient client = new WebClient();
    StringBuilder builder = new StringBuilder();

    for (int i = 2; i <= 10; i++)
    {
        try
        {
            this.tbStatus.Text = string.Format("Calling Server
[{0}]..... ", i);
            string currentCall = string.Format(Feed, i);
            string rss = await client.DownloadStringTaskAsync(new
Uri(currentCall));

            builder.Append(rss);
        }
        catch (Exception ex)
        {
            this.tbStatus.Text = string.Format("Error Occurred --
{0} for call :{1}, Trying next", ex.Message, i);
        }
        MessageBox.Show(string.Format("Downloaded Successfully!!!
Total Size : {0} chars.", builder.Length));
    }
}
```

In the preceding code, the async pattern is used. Notice the method returns a task reference. As I have already told you, a task is automatically returned after it encounters the first `await`. In the previous code, you must also notice that the variable `rss` is a string. If you check the return statement of the `DownloadStringAsync` method, you will notice that it is of `Task<string>` type, while when we place the contextual keyword, the type changes to just a string. Task being a representation of future when associated with `await`, will halt the execution until the right hand side has finished execution and the resultant is thrown to the variable defined on the line.



The await contextual keyword can only be associated with an awaitable pattern. In the .NET class library, it is declared as the naming convention of a method to suffix with Async when it is awaitable. For instance XXXAsync is awaitable while XXX is the normal method.

5. You can also call all the operations in parallel and use Task. WhenAll or the WhenAny API to wait for a single task. Modifying the AsynchronousCallServerMordernAsync method to support this behaviour is very easy.

```
public async Task AsynchronousCallServerMordernParallelAsync()
{
    List<Task<string>> lstTasks = new
List<Task<string>>();
    StringBuilder builder = new StringBuilder();
    for (int i = 2; i <= 10; i++)
    {
        using (WebClient client = new WebClient())
        {
            try
            {
                this.tbStatus.Text = string.
Format("Calling Server [{0}]..... ", i);
                string currentCall = string.Format(Feed,
i);
                Progress<int> p = new Progress<int>(v =>
prg.Value = v);
                Task<string> task = client.DownloadStringT
askAsync(currentCall, p);
                lstTasks.Add(task);
            }
            catch (Exception ex)
            {
                this.tbStatus.Text = string.Format("Error
Occurred -- {0} for call :{1}, Trying next", ex.Message, i);
            }
        }
    }
    string[] rss = await Task.WhenAll<string>(lstTasks);
    foreach (string s in rss)
        builder.Append(s);
    MessageBox.Show(string.Format("Downloaded
Successfully!!! Total Size : {0} chars.", builder.Length));
}
```

In the next button, we call the `DownloadStringTaskAsync` method directly and create `List<Task<string>>` from all the tasks returned from the method. We finally use `Task.WhenAll` to create an aggregation of all the task objects and retrieve the whole array of the downloaded string. Here all the web requests are made in parallel.

6. In the code in step 5, notice that we have used the `Progress` class to support progress of an operation on `WebClient`. The method `DownloadStringAsync` has an overload that takes `Progress` as an argument. (In current CTP release, we need to create an extension to do this). The progress takes a method to update the `Progress` notification which will be called when report of the progress is called.
7. As I have already told you, async methods do not create a thread when await is encountered. They actually stop the current progress and releases the thread with a task object returned. But this is not the case when you need to do some computation on something in the application. To do a CPU bound operation we use a separate thread to run the code in parallel inside `Task`.

```
await Task.Run(() =>
{
    //Switch to net Thread from ThreadPool
    result = this.DoCpuIntensiveWork(); //Very
CPU intensive
});
```

Here in the preceding code, `Task.Run` actually creates a new thread from `ThreadPool` under the hood and runs the action that has been passed to run a CPU intensive task. As a matter of fact, as all the `Task` object is awaitable, you can await on the task object received from the `Task.Run` method.

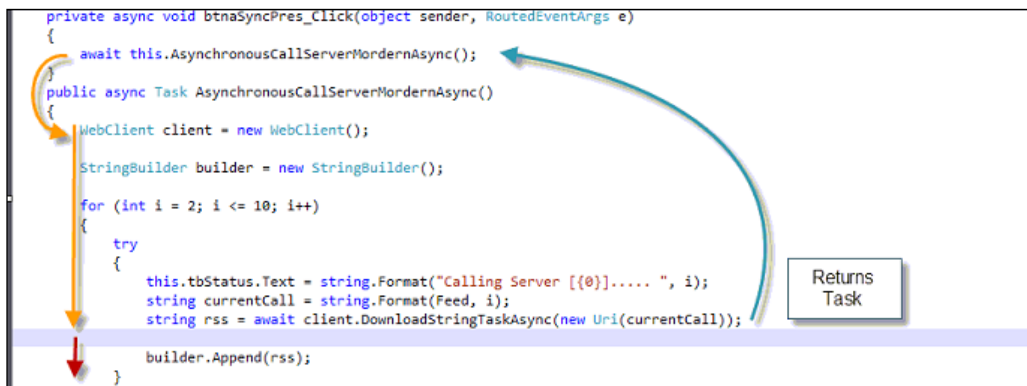
8. As every task supports `CancellationToken`, you can pass `CancellationToken` to most of the awaitable methods as well. You can use either return to immediately return when cancellation is requested or use `ThrowIfCancellationRequested` to throw an exception.

```
var cancelToken = new CancellationTokenSource();
var taskRequest = await DownloadTaskAsync(this.url, cancelToken.
Token)
```

In the preceding code `CancellationToken` is created using the `CancellationTokenSource` object and works in the same way as we saw earlier with tasks.

How it works...

It is a widely popular phrase that compilers are good at state machines. If you look at `IEnumerator` of C#, the compiler internally maintains a state-machine and stores every state of the enumeration inside it. The call to `yield` statement invokes the state machine and every enumeration automatically calls `MoveNext` of the enumeration to execute a certain set of instructions. The `yield` is just a syntactic sugar on the language which creates the actual `IEnumerator` and `IEnumerator` implementation on a separate class that also belongs to a state machine. This transformation of the code is already dealt inside the compiler itself to hide complexity around the state machine from the developer. C# `async` and `await` also works on the same model. The compiler rewrites the whole program during the compilation and builds and maintains the state machine inside a separate type. The C# asynchrony automatically identifies the thread and maintains a `SynchronizationContext` to avoid cross thread execution of messages. Hence, when you write `await` for a block, it actually yields the control after calling the method, and returns back a task object (that represents the ongoing method). If the ongoing method returns something, it is returned as `Task<T>`.



The code runs in two phases. First it calls the `AsynchronousCallServerMordern` method until it finds the first `await`, it then creates a task reference and registers the rest of the instructions inside a state machine as a method and returns the task object immediately. Hence, the UI will be available immediately after the first `await`. When the call to `DownloadStringTaskAsync` is finished, it automatically finds the context on the workflow and runs the existing code on the workflow. The state machine goes on executing the workflow steps divided based on the number of `await` statements and goes on executing them one by one. Therefore, if the `LongRunning` process is inside the line on which the users wait, the program will be free during its execution.

While you think of this situation, you might be thinking how is it possible to return the control before executing the whole method instance. When you are creating an `async` method, you internally create a state machine that keeps track of the whole execution of the method. If you have already used state machine before, you might already know that state machines are capable of keeping track of the current state of a flow. The `async` methods are nothing but an instance of a state machine created on the fly, which wraps the parameters, local variables, the state of the method that it is currently in, and so on. The method divides itself into a number of states based on the number of contextual `await` specified on the method. The flow gets executed until `await` is encountered and then everything is wrapped inside a call back on the next flow change.

For instance, let us consider the following code:

```
Console.WriteLine("Before await");
await TaskEx.Delay(1000);
Console.WriteLine("After await");
```

Now, when you compile the assembly, it actually creates an object of state machine with two methods (one for each state).

It takes the first delegate and associates it with the following:

```
Console.WriteLine("Before await");
await TaskEx.Delay(1000);
```

Say for instance, it names it as `State 1`. It takes the rest of the part to a new method.

```
Console.WriteLine("After await");
```

So after decomposition of the actual method into state machine, there are basically two methods each for one state. It executes the first part of the program synchronously and saves the state of the method into an object until it waits for a delay. `TaskAwaiter` automatically calls its call back when the delay is elapsed and executes the next state of the method. Finally when all the states are executed the result is stored into the task object and the method is turned into its completion.

There are quite a few things to remember in this regard, let's list them:

- ▶ For any `async` block it is important to have at least have one `await`, otherwise the whole block will execute synchronously.
- ▶ Any `async` method should always postfix `Async` (as a rule). This rule is just to differentiate a normal method with an `async` method. For instance, `MyMethodAsync` is an `async` method, while `MyMethod` is a normal method.
- ▶ Any `async` method can either return `void` (call and forget), `Task`, or `Task<T>` based on the `Result` the `await` method sends.
- ▶ The compiler does the adjustment to find the same caller and invokes the `GoTo` statement to execute the rest of the logic from the same position where `await` is invoked, rather than doing a `Callback`.

- ▶ Everything is managed by a state machine workflow by the compiler.
- ▶ CPU bound calls can have their own thread, but async-based mechanism does not necessarily mean creating a new thread while accessing any resource. For instance, it is necessary to block the calling thread while accessing a resource before the result arrives for network, or I/O bound calls do not require any new thread to be created under the hood.
- ▶ Any object which needs await must have `GetAwaiter` defined inside it.

Let us go a little deeper to understand what is happening under the hood. To check the compiled code, let us use **Reflector**.

As I have already explained, await works only for objects which implement the `GetAwaiter`. `Task` class in C# implements the method `GetAwaiter`, which returns another object (called `TaskAwaiter`) that is used to actually register the await pattern. Every `Awaiter` object should include some of the basic methods like `BeginAwait` and `EndAwait`. In BCL, there are a number of implementations of the awaitable methods, each of which defines its `awaiter`. If you look closely into `BeginAwait` and `EndAwait`, they are using the same technique that you might do in case of manually creating an asynchronous pattern block:

```
private bool TrySetContinuationForAwait(Action continuationAction)
{
    Action<Task> action2 = null;
    if (continuationAction == null)
    {
        throw new ArgumentNullException("continuationAction");
    }
    if (this.m_task == null)
    {
        throw new InvalidOperationException(SR.InvalidOperation_InstanceNotInitialized);
    }
    if (this.m_task.IsCompleted)
    {
        return false;
    }
    Action<Task> action = null;
    SynchronizationContext sc = this.m_flowContext ? SynchronizationContext.Current : null;
    if (sc != null)
    {
        if (action == null)
        {
            action = delegate (Task param0) {
                sc.Post(delegate (object state) {
                    ((Action) state)();
                }, continuationAction);
            };
        }
        this.m_task.ContinueWith(action, CancellationToken.None, TaskContinuationOptions.ExecuteSynchronously, TaskScheduler.Default);
    }
    else
    {
        if (action2 == null)
        {
            action2 = delegate (Task param0) {
                continuationAction();
            };
        }
        this.m_task.ContinueWith(action2, CancellationToken.None, TaskContinuationOptions.ExecuteSynchronously, this.m_flowContext ? Task
    }
    return true;
}
```

`BeginAwait` actually calls `TrySetContinuationForAwait`, which actually breaks apart the existing method body into two separate blocks and registers the next part of each `await` statement to the continuation of the task, just like the following:

```
Task1.ContinueWith(Task2);
```

Where, `Task2` represents the rest of the code to run in call back. So if you want your object to work with an asynchronous pattern, you must have `GetAwaiter` implemented on the type which returns an object that also implements `BeginAwait` and `EndAwait`.

There's more...

Async and **await** made things really easy. With the introduction of this new asynchronous pattern, it opens up a lot of new responsibilities. Let us take a look at a few more interesting facts about it.

How to write Async anonymous method or a lambda expression?

Delegates and lambda expressions are one of the most popular concepts which are rapidly adopted in .NET languages. Most of us today create short lambda expressions to pass to another method or call it directly from the code. Async delegates and lambda expressions need to be invented in the language before the release of original `async` method.

```
Func<Task> asyncDelegate = async delegate()
{
    await this.AsynchronousCallServerMordernAsync();
};
await asyncDelegate();
```

Here you can see the delegate actually returns a `Task` and we await on the `Task`.

You can also replace the preceding syntax to the following code:

```
Func<Task> asyncDelegate = async () =>
{
    await this.AsynchronousCallServerMordernAsync();
};
await asyncDelegate();
```

This code produces a lambda expression for the delegate. To write a delegate or a lambda expression which supports you to call contextual `await` inside it, you must precede the declaration with the `async` keyword.

You must also remember that you can cast the `asyncDelegate` to the `Action`, `Func<Task>`, or `Func<Task<T...>>` types.

Different awaitable methods

With the introduction of .NET `async` and `await` features, the base class library has also been modified to introduce newer awaitable methods inside it. The awaitable methods return either `Task` or `Task<T>` or `void`, and you can call them either from a normal function or an `async` method.

There are lot of `async` methods introduced inside the I/O classes, as stream has `ReadAsync`, `WriteAsync`, `FlushAsync`, `CopyToAsync`, and so on. The network APIs also introduce a lot of them. The main advantage of an awaitable method is that it does not need a new thread to block it when it can go without. The `async` methods generally use the `SynchronizationContext` class to hold itself inside an object and later when the message is available for the registered `SynchronizationContext`, it posts the data into it.

The `SemaphoreSlim` class also introduces the `WaitAsync` method that awaits an object before the lock is achieved. For instance:

```
private static SemaphoreSlim locker = new
    SemaphoreSlim(initialCount:1);
public static async Task DoWorkAsync()
{
    await locker.WaitAsync();
    try
    {
        ... // code here with awaits
    }
    finally { locker.Release(); }
}
```

In the preceding code the lock is achieved using the `async` pattern. The locker is awaited using the `WaitAsync` method and later when the method receives the lock, it tries to run the code and finally releases the locker. These types of methods come in really handy, as they do not require new threads to be created and blocked.

See also

- <http://bit.ly/AsyncPattern>

Working with Task Parallel Library data flows

Task Parallel Library data flows are recently introduced with .NET 4.5. The main objective of the library is to enhance the message passing techniques using the benefits of Task Parallel Library. It introduces a number of new classes and interfaces which deal with data flow patterns, and each of them forms special meaning and technique and to be used when required.

In most of the traditional systems of data flows, there are input and output on the two extreme ends of the pipeline. The data generally flows from multiple stages and processes the input to produce raw data which are used up by different processes. We post the messages to the system as input and wait for the processing to produce output. Dealing with such Message Passing techniques without using TPL Data Flows would not be easy as the foundation does not directly supports them. But with the introduction to the new framework, it becomes easy to build frameworks on top of it to support data flow blocks.

Getting ready

The approach of the TPL DataFlow are performed using data flow blocks. The TDF is based on actor-oriented models, and it does not guarantee isolation of one process with another. At its core, TPL Data Flow library is based mainly on two interfaces:

- ▶ `ISourceBlock<T>` which offers data source of the block
- ▶ `ITargetBlock<T>` which offers the target data of the blocks

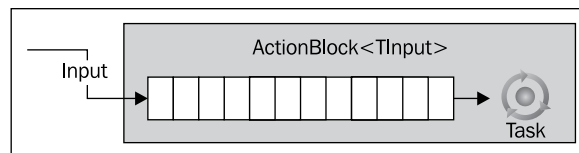
Each `DataFlowBlock` is inherited from `IDataFlowBlock` which has a source of type `ISourceBlock` and a target of type `ITargetBlock`. While it is passed through the block, the messages are processed. In this recipe we are going to cover all these TDF blocks which forms the entire library starting from the one of the most simplest construct to more and more complex construct. There are three types of blocks in TDF:

- ▶ **Executor Blocks:** Examples are `ActionBlock`, `TransformBlock`, and `TransformManyBlock`
- ▶ **Buffering Blocks:** Examples are `BufferBlock`, `BroadcastBlock`, and `WriteOnceBlock`
- ▶ **Joining Blocks:** Examples are `BatchBlock`, `JoinBlock`, and `BatchedJoinBlock`

How to do it...

To demonstrate the TPL data flow library, let us write some code and take example of the working principle of each of the types available on it.

1. `ActionBlock` is one of the most common and simplest data flow blocks that has been exposed from the library. `ActionBlock` takes a delegate to return the output.



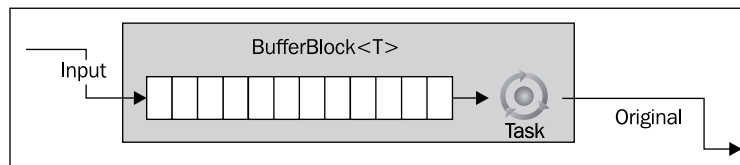
In the preceding figure, `ActionBlock` takes all the `Post` requests and performs the task (that means the method passed to it) and deletes it. To clear let us see the following code:

```
ActionBlock<int> ab = new ActionBlock<int>(v =>
{
    bool isEven = Compute(v);
    if(isEven)
        Console.WriteLine("The Number is Even");
    else
        Console.WriteLine("The number is odd");
});

ab.Post<int>(20);
ab.Post<int>(21);
```

This is the simplest type which takes a delegate and simply calls the method that has been passed as a delegate with a value that is sent. Here `ActionBlock<int>` is created, which makes it a delegate of `Action<int>`, hence we can post an integer value to it to get the delegate called.

2. `BufferBlock` is little different than that of `ActionBlock`. The `ActionBlock` takes a delegate and executes it one by one, but `BufferBlock` on the other hand provides storage of the posted message until it is processed such that both synchronous as well as asynchronous producer/consumer scenarios can be handled through it.



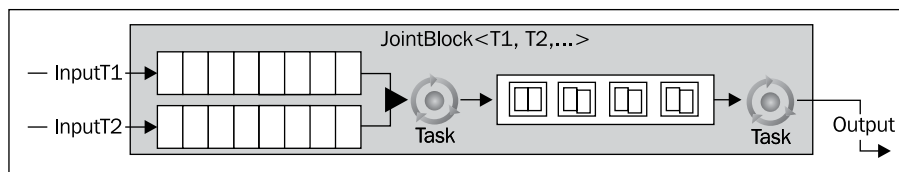
Here in the figure you can see that the task is performed on FIFO basis, such that the input is buffered right away when it is fed into the system as input and finally returns the output:

```
BufferBlock<string> buf = new BufferBlock<string>();
Task.Factory.StartNew(async () =>
{
    await buf.SendAsync<string>("http://www.abhisheksur.com");
    await buf.SendAsync<string>("http://sqlservergeeks.com");
});
Task.Factory.StartNew(async () =>
{
    while(true)
```

```
        Console.WriteLine(await buf.ReceiveAsync<string>());  
    });  
    Console.ReadLine();
```

Here in the preceding code we put two strings in the buffer using `SendAsync` and `ReceiveAsync`. As both support `async`, they are awaitable and hence, you can post both these messages at a time, and receive them from the buffer.

3. `JoinBlock` can group data coming from multiple data sources. `JoinBlock` exposes `TargetBlock` of input as properties based on the parameters and joins them into a tuple of all targets.



Here **InputT1** and **InputT2** are passed and the joined tuple taking one element from all the input produces the output:

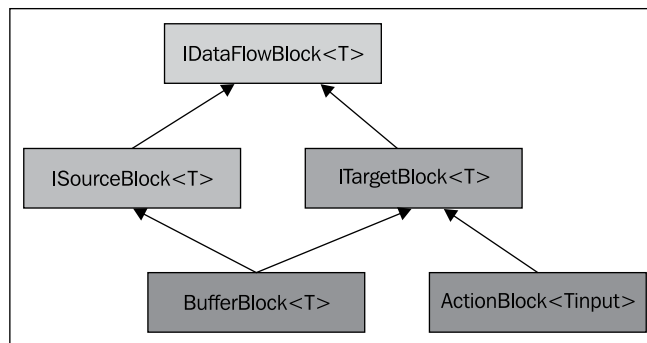
```
JoinBlock<string, double, int> jb = new JoinBlock<string, double,  
int>(new GroupingDataflowBlockOptions { Greedy = true });  
Task.Factory.StartNew(async () =>  
{  
    while (true)  
    {  
        string input1 = "Abhishek";  
        await jb.Target1.SendAsync(input1);  
        double input2 = 10.4d;  
        await jb.Target2.SendAsync(input2);  
        int input3 = 1950;  
        await jb.Target3.SendAsync(input3);  
    }  
});  
  
Task.Factory.StartNew(async () =>  
{  
    while (true)  
    {  
        var output = await jb.ReceiveAsync();  
        Console.WriteLine(output.ToString());  
    }  
});
```

In the preceding code `JoinBlock` receives three targets, each of which are exposed as properties. The three inputs are joined to produce one output. Remember, we have used greedy block. This ensures that it will allow `Receive` from the output only when all the inputs are ready.

How it works...

The TPL DataFlow blocks are really important when dealing with message processing. Each of the block is actually an implementation of `IDataFlowBlock<T>`. Each data flow blocks is designed to act as a data source using `ISourceBlock<T>`, and all data flow blocks that need to act as a target are implemented by `ITargetBlock<T>`.

Hence, the underlying data structure of every block is contained by three basic interfaces.



Here you can see the each data block can either have an `ISourceBlock`, `ITargetBlock`, or both, but they should have `IDataFlowBlock` to be considered as a DataFlow block. The `ActionBlock` does not implement `ISourceBlock` and hence cannot receive values.

There's more...

The TPL data flows are very important for consideration. We can use it for various purposes. Let us look at the configuration options available.

What are the configuration options available for TPL Data Flow blocks?

There are a lot of configuration options that may be applied to individual blocks. These are exposed through `DataflowBlockOptions` to the external world, and it has its derived types as `ExecutionDataflowBlockOptions` or `GroupingDataflowBlockOptions` depending on the type of the DataFlow block it is executing.

Let us look into a few of the important configuration options that are available to these blocks:

- ▶ **TaskScheduler:** This specifies how the task will schedule for the block. You can use either the default which is used by the TPL or you can implement your own and pass it to the block.
- ▶ **MaxDegreeOfParallelism:** This defines how many processing units will it create to run the tasks in parallel. Depending on the hardware you can restrict the parallelism of the tasks using this option.
- ▶ **MaxMessagePerTask:** You can configure the number of messages that each Task will process. This is important as there can be a huge number of messages posted on one end and if it creates that many tasks, it might compromise the overall performance. So there is a constant trade-off.
- ▶ **Greedy:** Greedy blocks are available to join blocks which group multiple targets into one to produce the output. If the block is configured as greedy, it will produce the output only when all data is available.
- ▶ **BoundedCapacity:** It specifies the maximum number of items that it can store in its buffer. You can restrict it for a block using configuration.
- ▶ **CancellationToken:** You can at any time pass `CancellationToken` to the block and input the necessary logic to cancel an operation.

How to link to one block to another for Target?

Each `ISourceBlock` allows linking to other blocks that support targets. There is a method `LinkTo` that allows you to pass the target to another block. For instance, `BufferBlock` stores the messages and later on can use `ActionBlock` which supports `ITargetBlock` to pass the message for processing. Let us consider this with an example:

```
BufferBlock<int> bb = new BufferBlock<int>();
ActionBlock<int> al = new ActionBlock<int>((a) =>
{
    Console.WriteLine("ActionBlock with value {0}", a);
})
);

bb.LinkTo(al);
Task t = new Task(() =>
{
    int i = 0;
    while (i < 10)
    {
        i++;
        bb.Post(i);
    }
});
t.Start();
```

So here rather than receiving from `BufferedBlock`, it links to `ActionBlock` for processing. The `LinkTo` method of `BufferBlock` redirects the target from it to `ActionBlock` that prints the value that has been passed. We pass 10 values to `BufferedBlock`, and if you run this code, you will see that all the messages are getting printed out using `ActionBlock`.

See also

- ▶ <http://bit.ly/TPLDataFlows>

4

Enhancements to ASP.NET

The goal of this chapter is to introduce the latest features of ASP.NET 4.5 and all the Visual Studio enhancements that make a richer development environment and a smarter end product. The chapter focuses on the latest programming trends, giving you insights on a few things that are interesting and yet important to learn before writing your program in ASP.NET:

- ▶ Understanding major performance boosters in ASP.NET web applications
- ▶ How to work with statically-typed model binding in ASP.NET applications
- ▶ Introduction to HTML5 and CSS3 in ASP.NET applications
- ▶ Working with jQuery in Visual Studio with ASP.NET
- ▶ Working with task-based asynchronous HttpHandler and HttpModules
- ▶ New enhancements to various Visual Studio editors

Introduction

Just like other technologies, ASP.NET had also come up with major alterations and advancements in recent times. Some of the things are merely related to actual development experiences rather than any benefits in terms of performance for the end user. But recent changes to ASP.NET have some real advantages and performance implication directly moving to the end users. The introduction of minification of JS and CSS files so easy now; anyone can now implement the same without using single lines of code. The code is inbuilt into the ASP.NET system and the API uses it gracefully to handle the release and debug environments.

It is not only restricted with the feature releases, but the Web has recently made a lot of advancements in terms of new HTML constructs appearing to the world. The adoption of HTML5 and CSS3 made the appearance of the Web almost identical to the desktop yet it runs on the browser. The browser is now capable of taking advantage of a graphics card or even multiple CPUs. There are APIs to directly communicate with the server from the client using a secured socket rather than the very old AJAX requests. The web look and feel has changed to support vector graphics, 2D and 3D canvas, WebGL, and so on, or even the browser supports local data storage, database, application cache, and much more. The world is moving towards a system where there will be a unified collaboration of online and offline activities.

This chapter focuses on some of the best introductions to the modern ASP.NET environment that catalyzes the modern web world. The chapter will guide you through the basic understanding of the concepts that will place you apart from other developers.

Understanding major performance boosters in ASP.NET web applications

Performance is one of the primary goals for any system. For a server, the throughput /time actually specifies the performance of the hardware or software in the system. It is important to increase performance and decrease the amount of hardware used for the throughput. There must be a balance between the two.

Performance is one of the key elements of web development. In the last phase of ASP.NET 4.5, performance was one of the key concerns for Microsoft. They made a few major changes to the ASP.NET system to make it more performant.

Performance comes directly, starting from the CPU utilization all the way back to the actual code you write. Each CPU cycle you consume for producing your response will affect performance. Consuming a large number of CPU cycles will lead you to add more and more CPUs to avoid site unavailability. As we are moving more and more towards the cloud system, performance is directly related to the cost. Here, CPU cycles costs money. Hence to make a more cost effective system running on the cloud, unnecessary CPU cycles should always be avoided.

.NET 4.5 addresses the problem to its core to support background GC which we will discuss in *Appendix, .NET Languages and its Construct*, with support for multicore JIT and so on. The background GC for the server introduces support for concurrent collection without blocking threads; hence the performance of the site is not compromised because of garbage collection as well. The multicore JIT in addition improves the start-up time of pages without additional work.

By the way, some of the improvements in technology can be really tangible to developers as well as end users. They can be categorized as follows:

- ▶ CPU and JIT improvements
- ▶ ASP.NET feature improvements

The first category is generally intangible while the second case is tangible. The CPU and JIT improvements, as we have already discussed, are actually related to server performance. JIT compilations are not tangible to the developers which means they will automatically work on the system rather than any code change while the second category is actually related to code. We will focus here mainly on the tangible improvements in this recipe.

Getting ready

To get started, let us start Visual Studio 2012 and create an ASP.NET project. If you are opening the project for the first time, you can choose the **ASP.NET Web Forms Application** project template. Visual Studio 2012 comes with a cool template which virtually creates the layout of a blank site. Just create the project and run it and you will be presented with a blank site with all the default behaviors you need. This is done without writing a single line of code.

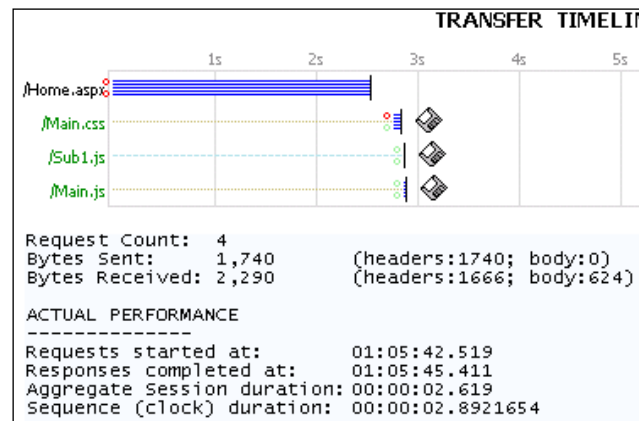
Now, if you look into **Solution Explorer**, the project is separated into folders, each representing its identification. For instance, the `Scripts` folder includes all the JavaScript associated with the site. You can also see the `Themes` folder in `Content`, which includes the CSS files. Generally, for production-level sites, we have large numbers of JavaScript and CSS files that are sometimes very big and they download to the client browser when the site is initially loaded. We specify the file path using the script or link path.

If you are familiar with web clients you will know that, the websites request these files in a separate request after getting the HTTP response for the page. As most browsers don't support parallel download, the download of each file adds up to the response. Even during the completion of each download there is a pause, which we call **network latency**. So, if you see the entire page response of a website, you will see that a large amount of response time is actually consumed by the download of these external files rather than the actual website response.

Let us create a page on the website and add few JavaScript files and see the response time using Fiddler:

#	Result	Protocol	Host	URL
1	200	HTTP	localhost:10984	/Home.aspx
2	200	HTTP	localhost:10984	/Resources/css/Main.css
3	200	HTTP	localhost:10984	/Resources/js/Sub1.js
4	200	HTTP	localhost:10984	/Resources/js/Main.js

The preceding screenshot shows how the browser requests the resources. Just notice, the first request is the actual request made by the client that takes half of a second, but the second half of that second is consumed by the requests made by the response from the server. The server responds with a number of CSS and JavaScript requests in the header, which have eventually been called to the same web server one by one. Sometimes, if the JavaScript is heavy, it takes a lot of time to load these individual files to the client which results in delay in response time for the web page. It is the same with images too. Even though external files are downloaded in separate streams, big images hurt the performance of the web page as well:



Here you can see that the source of the file contains the call to a number of files that corresponds to each request. When the HTML is processed on the browser, it invokes each of these file requests one by one and replaces the document with the reference of those files. As I have already told you, making more and more resources reduces the performance of a page. This huge number of requests makes the website very slow. The screenshot depicts the actual number of requests, the bytes sent and received, and the performance in seconds. If you look at some big applications, the performance of the page is reduced by a lot more than this.

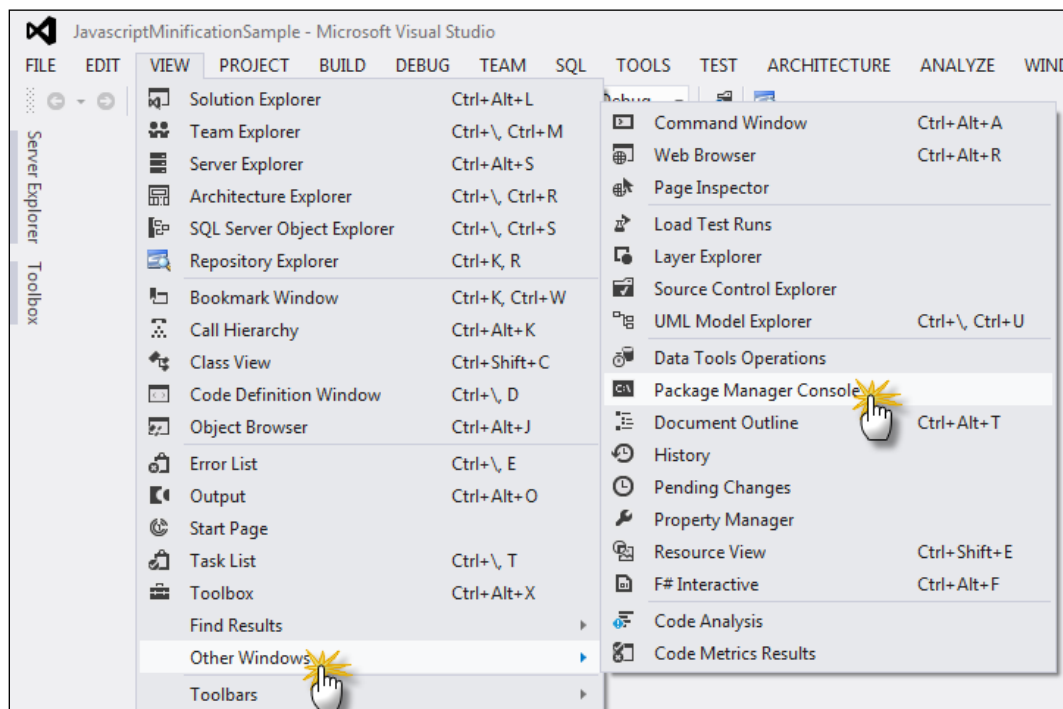
To address this problem we take the following two approaches:

- ▶ Minimizing the size of JavaScript and CSS by removing the whitespaces, newlines, tab spaces, and so on, or omitting out the unnecessary content
- ▶ Bundling all the files into one file of the same MIME type to reduce the requests made by the browser

ASP.NET addresses both of these problems and introduces a new feature that can both minimize the content of the JavaScript and CSS files as well as bundle all the JavaScript or CSS files together to produce one single file request from the site.

To use this feature you need to first install the package. Open Visual Studio 2012, select **View** | **Other Windows** | **Package Manager Console** as shown in the following screenshot.

Package Manager Console will open the PowerShell window for package management inside Visual Studio:



Once the package manager is loaded, type the following command:

```
Install-Package Microsoft.Web.Optimization
```

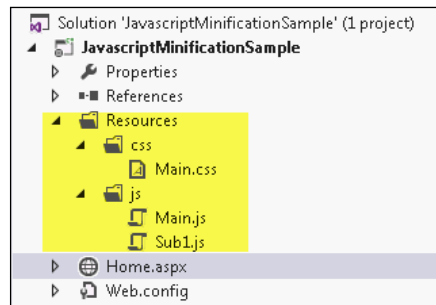
This will load the optimizations inside Visual Studio.

On the other hand, rather than opening **Package Manager Console**, you can also open **Nuget** package manager by right-clicking on the references folder of the project and select **Add Library Package Reference**. This will produce a nice dialog box to select and install the appropriate package.

In this recipe, we are going to cover how to take the benefits of bundling and minification of website contents in .NET 4.5.

How to do it...

1. In order to move ahead with the recipe, we will use a blank web solution instead of the template web solution that I have created just now. To do this, start Visual Studio and select the **ASP.NET Empty Web Application** template.
2. The project will be created without any pages but with a `web.config` file (a `web.config` file is similar to `app.config` but works on web environments).
3. Add a new page to the project and name it as `home.aspx`. Leave it as it is, and go ahead by adding a folder to the solution and naming it as `Resources`.
4. Inside `resources`, create two folders, one for JavaScript named `js` and another for stylesheets name `css`.
5. Create a few JavaScript files inside `js` folder and a few CSS files inside the `css` folder. Once you finish the folder structure will look like below :



Now let us add the files on the home page. Just drag-and-drop the `js` files one by one into the `head` section of the page. The page IDE will produce the appropriate tags for scripts and CSS automatically.

6. Now run the project. You will see that the CSS and the JavaScript are appropriately loaded. To check, try using Fiddler.
7. When you select the source of a page, you will see links that points to the JavaScript, CSS, or other resource files. These files are directly linked to the source and hence if we navigate to these files, it will show the raw content of the JavaScript.
8. Open Fiddler and refresh the page keeping it open in Internet Explorer in the debug mode. You will see that the browser invokes four requests. Three of them are for external files and one for the actual HTML file. The Fiddler shows how the timeframe of the request is maintained. The first request being for the `home.aspx` file while the others are automatically invoked by the browser to get the `js` and `css` files. You can also take a note at the total size of the whole combined request for the page.

9. Let's close the browser and remove references to the `js` and `css` files from the `head` tag, where you have dragged and added the following code to reference folders rather than individual files:

```
<script src="Resources/js" ></script>
<link rel="stylesheet" href="Resources/css" />
```

10. Open the `Global.asax` file (add if not already added) and write the following line in `Application_Start`:

```
void Application_Start(object sender, EventArgs e)
{
    //Adds the default behavior
    BundleTable.Bundles.EnableDefaultBundles();
}
```

Once the line has been added, you can now run the project and see the output.

11. If you now see the result in Fiddler, you will see all the files inside the scripts are clubbed into a single file and the whole file gets downloaded in a single request. If you have a large number of files, the bundling will show considerable performance gain for the web page.
12. Bundling is not the only performance gain that we have achieved using Optimization. Press `F5` to run the application and try to look into the actual file that has been downloaded as `js` and `css`. You will see that the bundle has been minified already by disregarding comments, blank spaces, new lines, and so on. Hence, the size of the bundles has also been reduced physically.
13. You can also add your custom `BundleTable` entries. Generally, we add them inside the `Application_Start` section of the `Global.asax` file, like so:

```
Bundle mybundle = new Bundle("~/mycustombundle",
    typeof(JsMinify));
mybundle.AddFile("~/Resources/Main.js");
mybundle.AddFile("~/Resources/Sub1.js");
mybundle.AddDirectory("/Resources/Files", "*.js", false);
BundleTable.Bundles.Add(mybundle);
```

The preceding code creates a new bundle for the application that can be referenced later on. We can use `AddFile` to add individual files to the `Bundle` or we can also use `AddDirectory` to specify a whole directory for a particular search pattern. The last argument for `AddDirectory` specifies whether it needs to search for a subdirectory or not. `JsMinify` is the default Rule processor for the JavaScript files. Similar to `JsMinify` is a class called `CssMinify` that acts as a default rule for CSS minification.

14. You can reference your custom bundle directly inside your page using the following directive:

```
<script src="mycustombundle" type="text/javascript" />
```

You will notice that the directive appropriately points to the custom bundle that has been created.

How it works...

Bundling and minification works with the introduction of the `System.Web.Optimization` namespace. `BundleTable` is a new class inside this namespace that keeps track of all the bundles that have been created in the solution. It maintains a list of all the `Bundle` objects, that is, list of JavaScript or CSS files, in a key-value pair collection. Once the request for a bundle is made, `HttpRequest` dynamically combines the files and/or directories associated with the bundle into a single file response.

Let us consider some other types that helps in transformation:

- ▶ `BundleResponse`: This class represents the response after the resources are bundled and minified. So `BundleResponse` keeps track of the actual response of the combined file.
- ▶ `IBundleTransform`: This type specifies the contract for transformation. Its main idea is to provide the transformation for a particular resource. `JsMinify` or `CssMinify` are the default implementations of `IBundleTransform`.
- ▶ `Bundle`: The class represents a resource bundle with a list of files or directories.

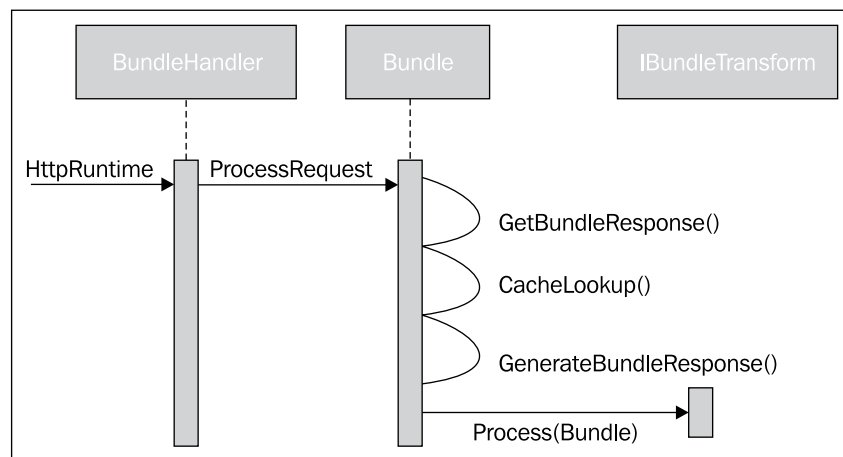
The `IBundleTransform` type specifies the rule for producing the `BundleResponse` class. To implement custom rules for a bundle, we need to implement this interface:

```
public class MyCustomTransform : IBundleTransform
{
    public void Process(BundleResponse bundleresponse)
    {
        // write logic to Bundle and minify...
    }
}
```

Here, the `BundleResponse` class is the actual response stream where we need to write the minified output to.

Basically, the application uses the default `BundleHandler` class to initiate the transform. `BundleHandler` is an `IHttpHandler` that uses `ProcessRequest` to get the response for the request from the browser. The process is summarized as follows:

- ▶ `HttpRuntime` calls the default `BundleHandler.ProcessRequest` method to handle the bundling and minification request initiated by the browser.
- ▶ `ProcessRequest` gets the appropriate bundle from the `BundleTable` class and calls `Bundle.ProcessRequest`.
- ▶ The `Bundle.ProcessRequest` method first retrieves the bundle's `Url` and invokes `Bundle.GetBundleResponse`.
- ▶ `GetBundleResponse` first performs a cache lookup. If there is no cache available, it calls `GenerateBundleResponse`.
- ▶ The `GenerateBundleResponse` method creates an instance of `BundleResponse`, sets the files to be processed in correct order, and finally invokes `IBundleTransform.Process`.
- ▶ The response is then written to `BundleResponse` from this method and the output is thrown back to the client.



The preceding flow diagram summarizes how the transformation is handled by ASP.NET. The final call to `IBundleTransform` returns the response back to the browser.

There's more...

Now let's talk about some other options, or possibly some pieces of general information that are relevant to this task.

How to configure the compilation of pages in ASP.NET websites

Compilation also plays a very vital role in the performance of websites. As we have already mentioned, we have background GC available to the servers with .NET 4.5 releases, which means when GC starts collecting unreferenced objects, there will be no suspension of executing threads on the server. The GC can start collecting in the background. The support of multicore JIT will increase the performance of non-JITed files as well.

By default, .NET 4.5 supports multicore JIT. If you want to disable this option, you can use the following code:

```
<system.web>
  <compilation profileGuidedOptimizations="None" />
</system.web>
```

This configuration will disable the support of spreading the JIT into multiple cores.

The server enables a **Prefetcher** technology, similar to what Windows uses, to reduce the disk read cost of paging during application startup. The Prefetcher is enabled by default, you can also disable this using the following code:

```
<system.web>
  <compilation enablePrefetchOptimization="false" />
</system.web>
```

This settings will disable the Prefetcher technology on the ASP.NET site. You can also configure your server to directly manipulate the amount of GC:

```
<runtime>
  <performanceScenario value="HighDensityWebHosting" />
```

The preceding configuration will make the website a high density website. This will reduce the amount of memory consumed per session.

What is unobtrusive validation

Validation plays a very vital role for any application that employs user input. We generally use ASP.NET data validators to specify validation for a particular control. The validation forms the basis of any input. People use validator controls available in ASP.NET (which include `RequiredFieldValidator`, `RangeValidator`, and so on) to validate the controls when either a page is submitted or when the control loses its focus or on any event the validator is associated with. Validators being most popular server-side controls that handles client-side validation by producing an inline JavaScript block inside the actual page that specifies each validator. Let us take an instance:

```
<asp:TextBox ID="Username" runat="server"></asp:TextBox>

<asp:RequiredFieldValidator
  ErrorMessage="Username is required!"
```

```

ControlToValidate="Username"
runat="server"></asp:RequiredFieldValidator>

<asp:RegularExpressionValidator
    ErrorMessage="Username can only contain letters!"
    ControlToValidate="Username"
    ValidationExpression="^[A-Za-z]+$"
    runat="server"></asp:RegularExpressionValidator>

```

The validator handles both the client-side and server-side validations. When the preceding lines are rendered in the browser, it produces a mess of inline JavaScript.

.NET 4.5 uses unobtrusive validation. That means the inline JavaScript is replaced by the data attributes in the HTML:

```

</div>
<div>
  <input name="username" type="text" id="username" />
  <span data-val-controltovalidate="username" data-val-errormessage="UserName&#32;is&#32;required" id="ctl02" data-
val="true" data-val-evaluationfunction="RequiredFieldValidatorEvaluateIsValid" data-val-initialvalue=""
style="visibility:hidden;">UserName is required</span>
  <span data-val-controltovalidate="username" data-val-errormessage="Username&#32;can&#32;only&#32;contain&#32;letters!"
id="ctl03" data-val="true" data-val-evaluationfunction="RegularExpressionValidatorEvaluateIsValid" data-val-
validationexpression="^[A-Za-z]+$" style="visibility:hidden;">username can only contain letters!</span>
</div>

```

This is a normal HTML-only code and hence performs better than the inline HTML and is also very understandable, neat, and clean.

You can also turn off the default behavior of the application just by adding the line in `Application_Start` of the `Global.asax` file:

```

void Application_Start(object sender, EventArgs e)
{
    //Disable UnobtrusiveValidation application wide
    ValidationSettings.UnobtrusiveValidationMode =
    UnobtrusiveValidationMode.None;
}

```

The preceding code will disable the feature for the application.

Applying appSettings configuration key values

Microsoft has implemented the ASP.NET web application engine in such a way that most of its configurations can be overridden by the developers while developing applications. There is a special configuration file named `Machine.config` that provides the default configuration of each of the config sections present for every application. `web.config` is specific to an application hosted on IIS. The IIS reads through the configuration of each directory to apply for the pages inside it.

As configuring a web application is such a basic thing for any application, there is always a need to have a template for a specific set of configuration without rewriting the whole section inside `web.config` again. There are some specific requirements from the developers perspective that could be easily customizable without changing too much on the config. ASP.NET 4.5 introduces magic strings that could be used as configuration key values in the `appSettings` element that could give special meaning to the configuration. For instance, if you want the default built-in JavaScript encoding to encode `&` character, you might use the following:

```
<appSettings>
    <add key="aspnet:JavaScriptDoNotEncodeAmpersand" value="false"
/>
</appSettings>
```

This will ensure that `&` character is encoded as `"\u0026"` which is the JavaScript-escaped form of that character. When the value is true, the default JavaScript string will not encode `&`.

On the other hand, if you need to allow `ScriptResource.axd` to serve arbitrary static files other than JavaScript, you can use another magic `appSettings` key to handle this:

```
<appSettings>
    <add key="aspnet:ScriptResourceAllowNonJsFiles" value="false" />
</appSettings>
```

The configuration will ensure that `ScriptResource.axd` will not serve any file other than the `.js` extension even if the web page has a markup `<asp:ScriptReference Path="~/myFile.txt" />`

Similar to this, you can also enable `UnobtrusiveValidationMode` on the website using a separate magic string on `appSetting` too:

```
<appSettings>
    <add key="ValidationSettings:UnobtrusiveValidationMode"
value="WebForms" />
</appSettings>
```

This configuration will make the application to render HTML5 data-attributes for validators.

There are a bunch of these `appSettings` key magic strings that you can use in your configuration to give special meaning to the web application. Refer to <http://bit.ly/ASPNETMagicStrings> for more information.

DLL intern in ASP.NET servers

Just like the reusability of string can be achieved using string intern tables, ASP.NET allows you to specify DLL intern that reduces the use of loading multiple DLLs into memory from different physical locations. The interning functionality introduced with ASP.NET reduces the RAM requirement and load time-even though the same DLL resides on multiple physical locations, they are loaded only once into the memory and the same memory is being shared by multiple processes. ASP.NET maintains symbolic links placed on the `bin` folder that map to a shared assembly. Sharing assemblies using a symbolic link requires a new tool named `aspnet_intern.exe` that lets you to create and manage the stored interned assemblies.

To make sure that the assemblies are interned, we need to run the following code on the source directory:

```
aspnet_intern -mode exec -sourcedir "Temporary ASP.NET files" -  
interndir "c:\assemblies"
```

This code will put the shared assemblies placed inside assemblies directory interned to the temporary ASP.NET files. Thus, once a DLL is loaded into memory, it will be shared by other requests.

See also

- Visit the following link:

<http://bit.ly/ASP45Performance1>

<http://bit.ly/ASP45Performance2>

How to work with statically-typed model binding in ASP.NET applications

Binding is a concept that attaches the source with the target such that when something is modified on the source, it automatically reflects to the target. The concept of binding is not new in the .NET framework. It was there from the beginning. On the server-side controls, when we set `DataSource`, we generally don't expect `DataSource` to automatically produce the output to be rendered onto the actual HTML. We expect to call a `DataBind` method corresponding to the control. Something magical happens in the background that generates the actual HTML from `DataSource` and produces the output. `DataSource` expects a collection of items where each of the items produces single entry on the control. For instance, if we pass a collection as the data source of a grid, the data bind will enumerate the collection and each entry in the collection will create a row of `DataGrid`. To evaluate each property from within the individual element, we use `DataBinder.Eval` that uses reflection to evaluate the contextual property with actual data.

Now we all know, `DataSource` actually works on a string equivalent of the actual property, and you cannot get the error before you actually run the page. In case of model binding, the bound object generates the actual object. Model binding does have the information about the actual object for which the collection is made and can give you options such as IntelliSense or other advanced Visual Studio options to work with the item.

Getting ready

`DataSource` is a property of the `DataBound` element that takes a collection and provides a mechanism to repeat its output replacing the contextual element of the collection with generated HTML. Each control generates HTML during the render phase of the ASP.NET page life cycle and returns the output to the client. The ASP.NET controls are built so elegantly that you can easily hook into its properties while the actual HTML is being created, and get the contextual controls that make up the HTML with the contextual data element as well. For a template control such as `Repeater`, each `ItemTemplate` property or the `AlternateItemTemplate` property exposes a data item in its callback when it is actually rendered. This is basically the contextual object of `DataSource` on the *n*th iteration. `DataSource.Eval` is a special API that evaluates a property from any object using Reflection. It is totally a runtime evaluation and hence cannot determine any mistakes on designer during compile time. The contextual object also doesn't have any type-related information inherent inside the control.

With ASP.NET 4.5, the `DataBound` controls expose the contextual object as generic types so that the contextual object is always strongly typed. The control exposes the `ItemType` property, which can also be used inside the HTML designer to specify the type of the contextual element. The object is determined automatically by the Visual Studio IDE and it produces proper IntelliSense and provides compile-time error checking on the type defined by the control.

In this recipe we are going to see step by step how to create a control that is bound to a model and define the HTML using its inherent `Item` object.

How to do it...

1. Open Visual Studio and start an **ASP.NET Web Application** project.
2. Create a class called `Customer` to actually implement the model. For simplicity, we are just using a class as our model:

```
public class Customer
{
    public string CustomerId { get; set; }
    public string Name { get; set; }
}
```

The `Customer` class has two properties, one that holds the identifier of the customer and another the name of the customer.

- Now let us add an ASPX file and add a Repeater control. The Repeater control has a property called `ModelType` that needs the actual logical path of the model class. Here we pass the customer.
- Once `ModelType` is set for the Repeater control, you can directly use the contextual object inside `ItemTemplate` just by specifying it with the `:Item` syntax:

```
<asp:Repeater runat="server" ID="rptCustomers"
ItemType="SampleBinding.Customer">
<ItemTemplate>
    <span><%# :Item.Name %></span>
</ItemTemplate>
</asp:Repeater>
```

Here in this Repeater control we have directly accessed the `Name` property from the `Customer` class. So here if we specify a list of `Customer` values to its data source, it will bind the contextual objects appropriately. The `ItemType` property is available to all `DataBound` controls.

- The `DataBound` controls in ASP.NET 4.5 also support CRUD operations. The controls such as `GridView`, `FormView`, and `DetailsView` expose properties to specify `SelectMethod`, `InsertMethod`, `UpdateMethod`, or `DeleteMethod`. These methods allow you to pass proper methods that in turn allow you to specify DML statements.
- Add a new page called `Details.aspx` and configure it as follows:

```
<asp:DetailsView SelectMethod="dvDepartments_GetItem"
ID="dvDepartments" UpdateMethod="dvDepartments_
UpdateItem" runat="server" InsertMethod="dvDepartments_
InsertItem" DeleteMethod="dvDepartments_DeleteItem"
ModelType="ModelBindingSample.ModelDepartment" AutoGenerateInsertB
utton="true">
</asp:DetailsView>
```

Here in the preceding code, you can see that I have specified all the DML methods. The code behind will have all the methods and you need to properly specify the methods for each operation.

How it works...

Every collection control loops through the `Datasource` control and renders the output. The `Bindable` controls support collection to be passed to it, so that it can make a template of itself by individually running the same code over and over again with a contextual object passed for an index. The contextual element is present during the phase of rendering the HTML. ASP.NET 4.5 comes with the feature that allows you to define the type of an individual item of the collection such that the template forces this conversion, and the contextual item is made available to the template.

In other words, what we have been doing with `Eval` before, can now be done easily using the `Item` contextual object, which is of same type as we define in the `ItemType` property. The designer enumerates properties into an IntelliSense menu just like a C# code window to write code easier.

Each databound control in ASP.NET 4.5 allows CRUD operations. For every CRUD operation there is a specific event handler that can be configured to handle operations defined inside the control. You should remember that after each of these operations, the control actually calls `DataBind` again so that the data gets refreshed.

There's more...

`ModelBinding` is not the only thing that is important. Let us discuss some of the other important concepts deemed fit to this category.

ModelBinding with filter operations

`ModelBinding` in ASP.NET 4.5 has been enhanced pretty much to support most of the operations that we regularly need with our ASP.NET pages. Among the interesting features is the support of filters in selection of control. Let us use `DetailsView` to introduce this feature:

```
<asp:DropDownList ID="ddlDepartmentNames" runat="server"
    ItemType="ModelBindingSample.ModelDepartment" AutoPostBack="true"
    DataValueField="DepartmentId" DataTextField="DepartmentName"
    SelectMethod="GetDepartments">
</asp:DropDownList>

<asp:DetailsView SelectMethod="dvDepartments_GetItems"
    ID="dvDepartments" UpdateMethod="dvDepartments_UpdateItem"
    runat="server" InsertMethod="dvDepartments_InsertItem"
    DeleteMethod="dvDepartments_DeleteItem"
    ItemType="ModelBindingSample.ModelCustomer" AutoGenerateIn
    sertButton="true">
</asp:DetailsView>
```

Here you can see the `DropDownList` control calls `Getdepartments` to generate the list of departments available. The `DetailsView` control on the other hand uses the `ModelCustomer` class to generate the customer list. `SelectMethod` allows you to bind the control with the data. Now to get the filter out of `SelectMethod` we use the following code:

```
public IQueryable<ModelCustomer> dvDepartments_GetItems([Control("ddlD
eartmentNames")]string deptid)
{
    // get customers for a specific id
}
```

This method will be automatically called when the drop-down list changes its value. The `departmentid` of the selected `DropDownItem` control is automatically passed into the method and the result is bound to `DetailsView` automatically. Remember, the `dvDepartments_GetItems` method always passes a `Nullable` parameter. So, if `departmentid` is declared as integer, it would have been passed as `int?` rather than `int`. The attribute on the argument specifies the control, which defines the value for the query element. You need to pass `IEnumerable` (`IQueryable` in our case) of the items to be bound to the control.

You can also specify a filter using `QueryString`. You can use the following code:



```
public IQueryable<Customer>
GetCustomers([QueryString]string departmentid)
{
    return null;
}
```

This code will take `departmentid` from the query string and load the `DataBound` control instead of the control specified within the page.

See also

Refer to the following links:

- ▶ <http://bit.ly/ASP45Mb>
- ▶ <http://bit.ly/ASP45MB>

Introduction to HTML5 and CSS3 in ASP.NET applications

Web is the media that runs over the Internet. It's a service that has already has us in its grasp. Literally, if you think of the Web, the first thing that can come into your mind is everything about HTML, CSS, and JavaScript. The browsers are the user agents that are used to communicate with the Web. The Web has been there for almost a decade and is used mostly to serve information about business, communities, social networks, and virtually everything that you can think of. For such a long period of time, users primarily use websites to see text-based content with minimum UI experiences and texts that can easily be consumed by search engines. In those websites, all that the browsers do is send a request for a page and the server serves the client with the appropriate page which is later rendered on the browser. But with the introduction to modern HTMLs, websites are gradually adopting interactivity in terms of CSS, AJAX, iFrame, or are even using sandboxed applications with the use of Silverlight, Flash, and so on.

Silverlight and Adobe AIR (Flash) are specifically likely to be used when the requirement is great interactivity and rich clients. They totally look like desktop applications and interact with the user as much as they can. But the problems with a sandboxed application are that they are very slow and need every browser to install the appropriate plugin before they can actually navigate to the application. They are heavyweight and are not rendered by the browser engine.

Even though they are so popular these days, most of the development still employs the traditional approach of HTML and CSS. Most businesses cannot afford the long loading waits or even as we move along to the lines of devices, most of these do not support them. The long term user requirements made it important to take the traditional HTML and CSS further, ornamenting it in such a way that these ongoing requirements could easily be solved using traditional code. The popularity of the ASP.NET technology also points to the popularity of HTML. Even though we are dealing with server-side controls (in case of ASP.NET applications), internally everything renders HTML to the browser.

HTML5, which was introduced by W3C and drafted in June 2004, is going to be standardized in 2014 making most of the things that need desktop or sandboxed plugins easily carried out using HTML, CSS, and JavaScript. The long term requirement to have offline web, data storage, hardware access, or even working with graphics and multimedia is easily possible with the help of the HTML5 technology. So basically what we had to rely on (the sandbox browser plugins) is now going to be standardized. In this recipe, we are going to cover some of the interesting HTML5 features that need special attention.

Getting ready

HTML5 does not need the installation of any special SDK to be used. Most of the current browsers already support HTML5 and all the new browsers are going to support most of these features. The official logo of HTML5 has been considered as follows:



HTML5 has introduced a lot of new advanced features but yet it also tries to simplify things that we commonly don't need to know but often need to remember in order to write code. For instance, the `DocType` element of an HTML5 document has been simplified to the following one line:

```
<!DOCTYPE html>
```

So, for an HTML5 document, the document type that specifies the page is simply HTML. Similar to `DocType`, the character set for the page is also defined in very simple terms:

```
<meta charset="utf-8" />
```

The character set can be of any type. Here we specified the document to be UTF – 8. You do not need to specify the `http-equiv` attribute or content to define `charset` for the page in an HTML5 document according to the specification. Let us now jot down some of the interesting HTML5 items that we are going to take on in this recipe. Semantic tags, better markups, descriptive link relations, micro-data elements, new form types and field types, CSS enhancements and JavaScript enhancements.



Not all browsers presently support every feature defined in HTML5. There are Modernizr scripts that can help as cross-browser polyfills for all browsers. You can read more information about it from the following link:
<https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills>

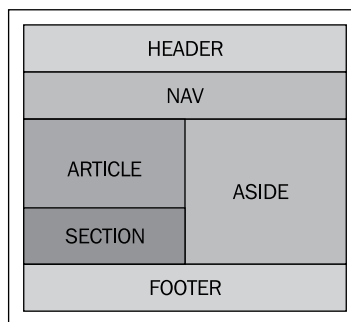
How to do it...

1. The HTML5 syntax has been adorned with a lot of important tags which include `header`, `nav`, `aside`, `figure`, and `footer` syntaxes that help in defining better semantic meaning of the document:

```
<body>
  <header>
    <hgroup>
      <h1>Page title</h1>
      <h2>Page subtitle</h2>
    </hgroup>
  </header>
  <nav>
    <ul>
      Specify navigation
    </ul>
  </nav>
  <section>
    <article>
```

```
<header>
  <h1>Title</h1>
</header>
<section>
  Content for the section
</section>
</article>
<article>
  <aside>
    Related links
  </aside>
  <figure>
    
    <figcaption>Special HTML5 Logo</figcaption>
  </figure>
  <footer>
    Copyright ©
    <time datetime="2010-11-08">2010</time>.
  </footer>
</body>
```

By reading the document, it clearly identifies the semantic meaning of the document. The `header` tag specifies the header information about the page. The `nav` tag defines the navigation panel. The `Section` tag is defined by articles and besides them, there are links. The `img` tag is adorned with the `Figure` tag and finally, the footer information is defined under the `footer` tag. A diagrammatic representation of the layout is shown as follows:



The vocabulary of the page that has been previously defined by `div` and CSS classes are now maintained by the HTML itself and the whole document forms a meaning to the reader.

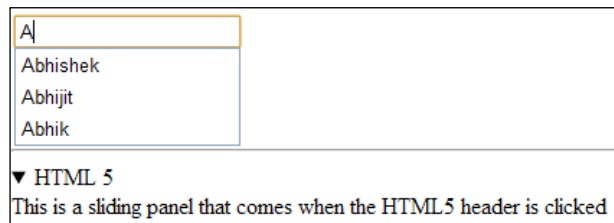
2. HTML5 not only improves the semantic meaning of the document, it also adds new markup. For instance, take a look at the following code:

```
<input list="options" type="text"/>
<datalist id="options">
  <option value="Abhishek"/>
  <option value="Abhijit"/>
  <option value="Abhik"/>
</datalist>
```

`datalist` specifies the autocomplete list for a control. A `datalist` item automatically pops up a menu while we type inside a textbox. The `input` tag specifies the list for autocomplete using the `list` attribute. Now if you start typing on the textbox, it specifies a list of items automatically:

```
<details>
  <summary>HTML 5</summary>
  This is a sliding panel that comes when the HTML5 header is
  clicked
</details>
```

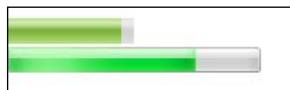
The preceding markup specifies a sliding panel container. We used to specify these using JavaScript, but now HTML5 comes with controls that handle these panels automatically:



3. HTML5 comes with a progress bar. It supports the `progress` and `meter` tags that define the progress bar inside an HTML document:

```
<meter min="0" max="100" low="40" high="90" optimum="100"
value="91">A+</meter>
<progress value="75" max="100">3/4 complete</progress>
```

The progress shows 75 percent filled in and the meter shows a value of 91:



4. HTML5 added a whole lot of new attributes to specify *aria* attributes and microdata for a block. For instance, consider the following code:

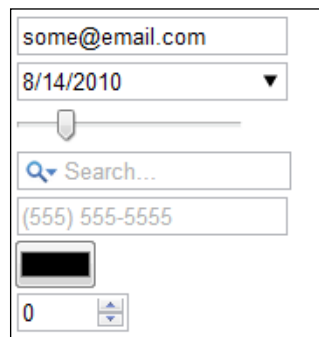
```
<div itemscope itemtype="http://example.org/band">
  <ul id="tv"
      role="tree"
      tabindex="0"
      aria-labelledby="node1">
    <li role="treeitem" tabindex="-1" aria-expanded="true">Inside
Node1</li>
  </li>
</ul>
```

Here, *Itemscope* defines the microdata and *ul* defines a tree with *aria* attributes. These data are helpful for different analyzers or even for automated tools or search engines about the document.

5. There are new Form types that have been introduced with HTML5:

```
<input type="email" value="some@email.com" />
<input type="date" min="2010-08-14" max="2011-08-14"
value="2010-08-14"/>
<input type="range" min="0" max="50" value="10" />
<input type="search" results="10" placeholder="Search..." />
<input type="tel" placeholder="(555) 555-5555"
      pattern="^\(?\d{3}\)?[-\s]\d{3}[-\s]\d{4}.*?$" />
<input type="color" placeholder="e.g. #bbbbbb" />
<input type="number" step="1" min="-5" max="10" value="0" />
```

These inputs types give a special meaning to the form:



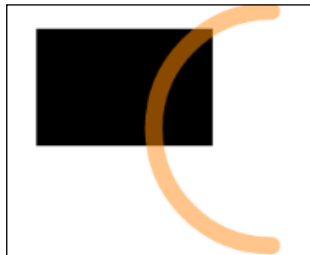
The preceding figure shows how the new controls are laid out when placed inside a HTML document. The controls are email, date, range, search, tel, color, and number respectively.

6. HTML5 supports vector drawing over the document. We can use a canvas to draw 2D as well as 3D graphics over the HTML document:

```
<script>
    var canvasContext = document.getElementById("canvas").
    getContext("2d");
    canvasContext.fillRect(250, 25, 150, 100);

    canvasContext.beginPath();
    canvasContext.arc(450, 110, 100, Math.PI * 1/2, Math.PI * 3/2);
    canvasContext.lineWidth = 15;
    canvasContext.lineCap = 'round';
    canvasContext.strokeStyle = 'rgba(255, 127, 0, 0.5)';
    canvasContext.stroke();
</script>
```

Consider the following diagram:



The preceding code creates an arc on the canvas and a rectangle filled with the color black as shown in the diagram. The canvas gives us the options to draw any shape within it using simple JavaScript.

7. As the world is moving towards multimedia, HTML5 introduces `audio` and `video` tags that allow us to run audio and video inside the browser. We do not need any third-party library or plugin to run audio or video inside a browser:

```
<audio id="audio" src="sound.mp3" controls></audio>
<video id="video" src="movie.webm" autoplay controls></video>
```

The `audio` tag runs the audio and the `video` tag runs the video inside the browser. When controls are specified, the player provides superior browser controls to the user.

8. With CSS3 on the way, CSS has been improved greatly to enhance the HTML document styles. For instance, CSS constructs such as `.row:nth-child(even)` gives the programmer control to deal with a particular set of items on the document and the programmer gains more granular programmatic approach using CSS.

How it works...

HTML5 is the standardization to the web environments with W3C standards. The HTML5 specifications are still in the draft stage (a 900-page specification available at <http://www.w3.org/html/wg/drafts/html/master/>), but most modern browsers have already started supporting the features mentioned in the specifications. The standardization is due in 2014 and by then all browsers need to support HTML5 constructs.

Moreover, with the evolution of smart devices, mobile browsers are also getting support for HTML5 syntaxes. Most smart devices such as Android, iPhone, or Windows Phone now support HTML5 browsers and the HTML that runs over big devices can still show the content on those small browsers.

HTML5 improves the richness of the web applications and hence most people have already started shifting their websites to the future of the Web.

There's more...

HTML5 has introduced a lot of new enhancements which cannot be completed using one single recipe. Let us look into some more enhancements of HTML5, which are important to know.

How to work with web workers in HTML5

Web workers are one of the most awaited features of the entire HTML5 specification. Generally, if we think of the current environment, it is actually turning towards multicore machines. Today, it's verbal that every computer has at least two cores installed in their machine. Browsers are well capable of producing multiple threads that can run in parallel in different cores. But programmatically, we cannot have the flexibility in JavaScript to run parallel tasks in different cores yet.

Previously, developers used `setTimeout`, `setInterval`, or `XMLHttpRequest` to actually create non-blocking calls. But these are not truly a concurrency. I mean, if you still put a long loop inside `setTimeout`, it will still hang the UI. Actually these works asynchronously take some of the UI threads time slices but they do not actually spawn a new thread to run the code.

As the world is moving towards client-side, rich user interfaces, we are prone to develop codes that are capable of computation on the client side itself. So going through the line, it is important that the browsers support multiple cores to be used up while executing a JavaScript.

Web workers are actually a JavaScript type that enable you to create multiple cores and run your JavaScript in a separate thread altogether, and communicate the UI thread using messages in a similar way as we do for other languages.

Let's look into the code to see how it works:

```
var worker = new Worker('task.js');
worker.onmessage = function(event) { alert(event.data); };
worker.postMessage('data');
```

Here we will load `task.js` from `Worker`. `Worker` is a type that invokes the code inside a JavaScript in a new thread. The start of the thread is called using `postMessage` on the worker type. Now we have already added a callback to the event `onmessage` such that when the JavaScript inside `task.js` invokes `postMessage`, this message is received by this callback. Inside `task.js` we wrote:

```
self.onmessage = function(event) {
// Do some CPU intensive work.
self.postMessage("recv'd: " + event.data);
};
```

Here after some CPU-intensive work, we use `self.postMessage` to send the data we received from the UI thread and the `onmessage` event handler gets executed with message received data.

Working with Socket using HTML5

HTML5 supports full-duplex bidirectional sockets that run over the Web. The browsers are capable of invoking socket requests directly using HTML5. The important thing that you should note with sockets is that it sends only the data without the overload of HTTP headers and other HTTP elements that are associated with any requests. The bandwidth using sockets is dramatically reduced. To use sockets from the browsers, a new protocol has been specified by W3C as a part of the HTML5 specification. `WebSocket` is a new protocol that supports two-way communication between the client and the server in a single TCP channel.

To start socket server, we are going to use `node.js` for server side. Install `node.js` on the server side using the installer available at <http://nodejs.org/dist/v0.6.6/node-v0.6.6.msi>. Once you have installed `node.js`, start a server implementation of the socket:

```
var io = require('socket.io');
//Creates a HTTP Server
var socket = io.listen(8124);
//Bind the Connection Event
//This is called during connection
socket.sockets.on('connection',function(socket){
//This will be fired when data is received from client
  socket.on('message', function(msg){
    console.log('Received from client ',msg);
  });
//Emit a message to client
  socket.emit('greet',{hello: 'world'});
```



```
//This will fire when the client has disconnected
socket.on('disconnect', function(){
    console.log('Server has disconnected');
});
});
```

In the preceding code, the server implementation has been made. The `require('socket.io')` code snippet specifies the include module header. `socket.io` provides all the APIs from `node.js` that are useful for socket implementation. The `Connection` event is fired on the server when any client connects with the server. We have used to listen at the port 8124 in the server. `socket.emit` specifies the emit statement or the response from the server when any message is received by it. Here during the `greet` event, we pass a JSON object to the client which has a property `hello`. And finally, the `disconnect` event is called when the client disconnects the socket.

Now to run this client implementation, we need to create a HTML file:

```
<html>
  <title>WebSocket Client Demo</title>
  <script src="http://localhost:8124/socket.io/socket.io.js"></script>
  <script>
    //Create a socket and connect to the server
    var socket = io.connect('http://localhost:8124/');
    socket.on("connect",function(){
      alert("Client has connected to the server");
    });
    socket.on('greet', function (data) {
      alert(data.hello);
    }
  );
</script>
</html>
```

Here we connect the server at 8124 port. The `connect` event is invoked first. We call an `alert` method when the client connects to the server. Finally, we also use the `greet` event to pass data from the server to the client. Here, if we run both the server and the client, we will see two alerts; one when the connection is made and the other alert to greet. The `greet` message passes a JSON object that greets with `world`.

The URL for the socket from the browser looks like so:

```
[scheme] '://' [host] '/' [namespace] '/' [protocol version] '/'
[transport id] '/' [session id] '/' ( '?' [query] )
```

Here, we see:

- ▶ `Scheme`: This can bear values such as `http` or `https` (for web sockets, the browser changes it to `ws://` after the connection is established, it's an upgrade request)
- ▶ `host`: This is the host name of the socket server
- ▶ `namespace`: This is the `Socket.IO` namespace, the default being `socket.io`
- ▶ `protocol version`: The client support default is 1
- ▶ `transport id`: This is for the different supported transports which includes `WebSockets`, `xhr-polling`, and so on
- ▶ `session id`: This is the web socket session's unique session ID for the client

Getting GeoLocation from the browser using HTML5

As we are getting inclined more and more towards devices, browsers are trying to do their best to actually implement features to suit these needs. HTML5 introduces `GeoLocation` APIs to the browser that enable you to get the location of your position directly using JavaScript.

In spite of it being very much primitive, browsers are capable of detecting the actual location of the user using either Wi-Fi, satellite, or other external sources if available. As a programmer, you just need to call the location API and everything is handled automatically by the browser.

As `geolocation` bears sensitive information, it is important to ask the user for permission. Let's look at the following code:

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(function(position) {
    var latLng = "[" + position.coords.latitude + "," + position.coords.
      longitude + "with accuracy: " + position.coords.accuracy;
    alert(latLng);
  }, errorHandler);
}
```

Here in the code we first detect whether the `geolocation` API is available to the current browser. If it is available, we can use `getCurrentPosition` to get the location of the current device and the accuracy of the position as well.

We can also use `navigator.geolocation.watchPosition` to continue watching the device location at an interval when the device is moving from one place to another.

Working with local IndexedDB storage in a browser using HTML5

`IndexedDB` is another important storage mechanism in the browser industry along with HTML5. This is a high-speed data access key-value collection. On November 18, 2010, W3C announced that they are going to deprecate the addition to SQL database but rather they will go with the `IndexedDB` specification in HTML5.

Let's now look into the code on how to work with IndexedDB:

```
if ('webkitIndexedDB' in window) {
    window.indexedDB = window.webkitIndexedDB;
    window.IDBTransaction = window.webkitIDBTransaction;
}
else if ('moz_indexedDB' in window) {
    window.indexedDB = window.moz_indexedDB;
}

if (window.indexedDB) {
    idbRequest_ = window.indexedDB.open("Test", "A test object store.");
    idbRequest_.onerror = idbError_;
    idbRequest_.addEventListener('success', function(event) {
        idb_ = event.srcElement.result;
        idbShow_(event);
    }, false);
}
```

In the preceding code, we first checked whether `indexedDB` is available for the browser where the code is running. As the specification is not yet ready in most browsers, as a developer we need to pay special attention to checking whether `indexedDB` is available before using it. Once we get the object we call its `open` method to open a test database. Here we call it `test`. We can also put a small description about the database.

To store the data object we need to use the following code:

```
var db = window.indexedDB;
var trans = db.transaction(["abhishek"], IDBTransaction.READ_WRITE,
    0);
var store = trans.objectStore("abhishek");
var request = store.put({
    "text": 'Hi, this is Abhishek',
    "timeStamp" : new Date().getTime()
});

request.onsuccess = function(e) {
    // Re-render all the data
};

request.onerror = function(e) {
    // show e.value };
};
```

In the preceding code snippet, we tried to put a JSON object inside the `indexedDB` storage of a browser using the `READ_WRITE` mode access. To do this, we created a store and named it `Abhishek`. We put some JSON-formatted data into it. Similar to all storage, it also takes two callbacks for success and error.

Now to retrieve the data from the store we use the following code:

```
var db = window.indexedDB;
var trans = db.transaction(["abhishek"], IDBTransaction.READ_WRITE,
0);
var store = trans.objectStore("abhishek");

// Get everything in the store;
var keyRange = IDBKeyRange.lowerBound(0);
var cursorRequest = store.openCursor(keyRange);

cursorRequest.onsuccess = function(e) {
var result = e.target.result;
if(!result == false)
return;

//write result.value
result.continue();
};

cursorRequest.onerror = function(e){
<pre>    };
};
```

So here the data is retrieved using a cursor and fetching all the data one by one. The resulting object will hold the actual data that has been stored inside `indexedDb`.

In the same way as mentioned, you can use `var request = store.delete(id);` to delete an element from object store.

It is sometimes important to detect the actual quota of storage that the current client browser supports before actually storing data into it. To prevent the `QUOTA_EXCEEDED_ERR` exception from happening in the browser. As a developer you will always need to query which memory can be stored without exceeding the quota given by the client to the application.

Let us take a look at how to query the local storage:

```
webkitStorageInfo.queryUsageAndQuota( webkitStorageInfo.TEMPORARY,
showData, showerror);
webkitStorageInfo.queryUsageAndQuota( webkitStorageInfo.PERSISTENT,
showData, showerror);
```

These two lines of code will show the `queryUsageAndQuota` value of the browser. If the data cannot be retrieved, it will call the error callback else it will call `showData`:

```
function showData (used, remaining) {  
    /// the used and remaining bytes will be received inside this method  
    automatically.  
}
```

As far as I know, Mozilla Firefox, Chrome, Opera supports 5 MB of storage while Internet Explorer supports 10 MB of local storage, but this data might change.

Using application cache for a site in HTML5

Well, yet another offline storage can be the actual HTML that has been transferred from the server. It is an important fact that even if we use browser cache using the cache header, the browser does not respond properly when offline, as it does when one is online. HTML5 introduces a new way of programming which ensures that the browser cache can be used up to cache individual pages of the website which work in the same way even when offline than when it gets online.

To use application cache, we first need to understand that it works with a manifest file. Let us consider an example:

```
<html manifest="cache.appcache">
```

Here in this code, the HTML points to the cache file. We generally give the extension as `appcache` to the external header. The content of this file is a plain text and will hold all the files that need to be used as application cache.

The cache file contains the following code:

```
CACHE MANIFEST  
# version 1.0.0  
  
CACHE:  
/html5/src/logic.js  
/html5/src/style.css  
/html5/src/background.png  
  
NETWORK:  
*
```

Here we have specified the content of the manifest that needs to be cached. Here, the `background.png`, `style.css`, and `logic.js` will remain cached and will act normally to the browser as if the application is online. The `network *` specifies all other files that needed to be online. You can also define the fallback for the cached manifest:

```

window.applicationCache.addEventListener('updateready', function(e) {

    if (window.applicationCache.status == window.applicationCache.
    UPDATEREADY) {
        window.applicationCache.swapCache();
        if (confirm('A new version of this site is available. Load it?')) {
            window.location.reload();
        }
    }
}, false);

```

So in the preceding code we add an event handler to the `UpdateReady` event such that when the cache update is ready it will show a dialog to reload the page automatically.

Detecting online status of the browser using HTML5

Since HTML5 has been introduced, lately there has been a great amount of thrust on the web browser market to cope with different APIs built on top of it, thus producing a lot of interesting features. HTML5 tries to eradicate the use of third-party plugins completely by giving the best out of it. Web applications can now run without even having an active connection to the Internet. They can store offline data into their storage and later when the machine is turned online, that data can be silently uploaded to the server. When doing this, it is important to determine when the application is online and when it isn't. Let us try to detect the browser online status:

```

var addEvent = (function () {
    if (document.addEventListener) {
        return function (el, type, fn) {
            if (el && el.nodeName || el === window) {
                el.addEventListener(type, fn, false);
            } else if (el && el.length) {
                for (var i = 0; i < el.length; i++) {
                    addEvent(el[i], type, fn);
                }
            }
        };
    } else {
        return function (el, type, fn) {
            if (el && el.nodeName || el === window) {
                el.attachEvent('on' + type, function () { return fn.call(el,
                window.event); });
            }
        };
    }
})();

```

```
    } else if (el && el.length) {  
        for (var i = 0; i < el.length; i++) {  
            addEvent(el[i], type, fn);  
        }  
    }  
};  
}  
})();
```

You should already know that there are browser compatibility issues as most of the browsers still lack standardization. `document.addEventListener` works in most of the browsers except IE. So to handle this, we have to bypass the availability of the event handler.

Now we will subscribe to the event we call:

```
addEvent(window, 'online', online);  
addEvent(window, 'offline', online);  
online({ type: 'ready' });
```

So basically here we trap the online and offline events of the `Window` object using either `attachEvent` or `addEventListener` to show the online or offline status. The online status is the event callback. Let's take a look on the event handler:

```
function online(event) {  
    document.getElementById('status').innerHTML = navigator.onLine ?  
    'online' : 'offline';  
}
```

The preceding code determines the online status of the browser using `navigator.onLine`, it returns true when the client is online or else false.

Working with notifications in browsers using HTML5

Notification is one of the interesting things that browsers are adding support to. Generally, when we think of a web notification, we always go for some HTML pop up or use a new window through JavaScript. But those HTML generally do not follow any standards and even look different to the user at different sites. Hence, a few of the notifications lack consistency. HTML5 introduces new notification specifications that enables the browser to send its own notification rather than going with custom notifications from the developer.

Notifications are now currently implemented in Chrome, but it will be implemented in the latest releases of other browsers as well. Let us look how to use notifications in your browser.

To request for notifications, use the following code:

```
window.webkitNotifications.requestPermission();
```

When the browser asks for notifications, it will pop up one message to the user to allow or deny. If the user allows the notification, the notification service gets enabled and the site can then send notifications to the browser:

```
if (window.webkitNotifications.checkPermission() == 0) {  
  var notification = window.webkitNotifications.  
    createNotification(imgpath, 'Notification received', 'Hii, this is  
    special notification');  
  notification.show();  
}
```

So when we use `createNotification`, it will create a notification object with the image, title, and the message which needs to be notified. The `show()` method will show the notification to the user.

See also

- ▶ Refer to the following link:

<http://bit.ly/html5-Intro>

Working with jQuery in Visual Studio with ASP.NET

Today, jQuery has been adopted by most (if not all) of the big giants in the browser industry and hence the requirement of knowing jQuery has also been huge. Microsoft has adopted jQuery and has made Visual Studio to natively support jQuery with all its syntaxes directly within the IDE. This made writing JavaScript in a jQuery syntax easier than ever before working with Visual Studio. In this recipe, I am going to give you some basic ideas about working with jQuery and will guide you through how to make use of it in your day-to-day programming.

Getting ready

jQuery is a widely-accepted JavaScript library that helps you to define your markups on the client side of a web page. It is a lightweight "write less, do more" JavaScript library. jQuery has the following few features:

- ▶ HTML element selectors
- ▶ HTML element manipulators
- ▶ CSS manipulators
- ▶ JavaScript effects and animations
- ▶ DOM traversal

- ▶ AJAX
- ▶ HTML event functions
- ▶ Utilities

The library is free to be used commercially as well. So you can download the library from `jquery.com` and add it to your web page. ASP.NET templates automatically add the JavaScript library for you, so if you are using Visual Studio, you do not need to download jQuery separately.

jQuery can be downloaded into two formats (if you are using Visual Studio 2012, jQuery will be automatically added to the solution under the `scripts` folder). The first one which is the entire jQuery source code, used during debugging your application or during development environment. Another version, which is suffixed with `-min`, is the minified version of the library and it is considerably smaller in size and should be used in production environments. You can also refer the library directly from the Web using the Microsoft or Google server. The links are as follows:

<http://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js>

<http://ajax.microsoft.com/ajax/jquery/jquery-1.6.2.min.js>

The best and the most popular jQuery construct is `$`. `$` is defined as a function that acts as a primary selector for the DOM elements. It is important to note that DOM elements are completely available only when the application is ready. So if you want to run some code just after your document is fully loaded, you can write it like this:

```
$(document).ready(function() {  
    /// Write code that need to run when document is fully loaded.  
});
```

Rather than hooking up inside the DOM element for the `ready` method, jQuery gives you a great construct to bind a JavaScript method directly to the `ready` event of the document using the preceding syntax. We are going to use the following HTML to inspect our page using jQuery:

```
<p class="intro">  
    Hi, Welcome to JQuery Tutorial</p>  
<p>  
    I am Abhishek Sur, living in India</p>  
<p>  
    My friend is Abhijit who lives in India too.</p>  
<div id="msges">  
</div>  
Who is your favorite:  
<ul id="choose">  
    <li>
```

```

        <input type="checkbox" data="sg" />Scott Gutherine</li>
    </li>
    <input type="checkbox" data="sh" />Scott Hansleman</li>
    </li>
    <input type="checkbox" data="jp" />John Papa</li>
</ul>
<div id="ajaxResponse">
</div>
<input id="btnSelectors" type="button" value="InvokeSelectors" />
<input id="btnslideToggle" type="button" value="Invoke Silde" />
<input id="btnAppend" type="button" value="Append text" />
<input id="btnAnimate" type="button" value="Animate Ps" />
<input id="btnAjax" type="button" value="Call Ajax" />

```

Now let us move ahead step by step with the recipe.

How to do it...

1. jQuery selectors play a vital role in working with HTML elements. \$ is used as a selector for DOM elements. Let us consider the following JavaScript code:

```

var allelements = $("*"); //Returns all elements of the page
var ullist = $("#choose"); //Returns the unordered list with id
choose.
var firstP = $("p:first"); // gets the first P element
var lastP = $("p:last"); // gets the last P element
var pOdd = $("p:odd"); // gets list of all odd P
var peven = $("p:even"); //gets list of all even P
var fromClass = $(".intro"); // gets list of all elements that has
intro class applied
var secondli = $("ul li:eq(1)"); // gets li whose index is 1
var thirdli = $("ul li:gt(1)"); // gets li whose index greater
than 1
var getElementswithIndia = $("p:contains('India')"); // gets all P
that has India
var checkedItems = $("input:checked"); // Finds all checked inputs
var findbycustomattribute = $("[data=sh]"); //finds element that
has sh in data attribute
var findbydata = $("[data]"); // returns all elements that has
data attribute
//All selectors returns array.
findbydata.hide(); //hides all element that returned from the
selector

```

Here we have shown you a few selectors which you can use in your jQuery to get elements from the document. You should note that each selector actually returns a jQuery object that has the actual DOM elements enumerated. The aggregate methods which include `hide()`, `show()`, `val()`, and so on, when applied on an object that has multiple objects selected, the methods will apply on all those objects as well. In other words, `findbydata` here selects all the checkboxes on the document and `hide()` will hide all the checkboxes that are selected.

2. jQuery being very good at its selectors or with existing DOM elements, is also capable of changing the DOM elements dynamically. Unlike the way of JavaScript `createElement` or `appendChild`, it has a few superior APIs available that makes the HTML DOM manipulation very easy. For instance, consider the following code:

```
$("#msges").append("Add text after existing. <hr/>");  
$("#msges").prepend("Add text before existing. <hr/>");  
$("#msges").html("Replaces the text. <hr/>");
```

This code adds or replaces the text to all elements that have been selected by the selector. Similarly, you can also call `remove` or `detach` to remove content from the DOM element.

3. CSS manipulation using jQuery has also been very easy. There are APIs to add and remove CSS classes:

```
$("#msges").css("background-color", "yellow");  
$("#msges").css({"background-color":"yellow", "font-size":  
"50px"});  
$("#msges").height("300px");  
$("#msges").addClass("mark"); //adds the class mark  
$("#msges").removeClass("mark");
```

Here you can see how easy it is to manipulate CSS with jQuery. You can add CSS to the selected elements, specify the height/width, or use the custom CSS class to add a class or remove a class to the element.

4. DOM elements raise events whenever some action is performed on the HTML. For instance, when a button is clicked or the mouse is moved over some object, some data changes on a textbox or some other control. Most of the times to deal with these events we need to hook event handlers. jQuery provides superior API support to deal with these events. For instance, consider the following code:

```
$("#btnSelectors").click(callSelectors);  
$("#btnslideToggle").bind("click", function () {  
    $("p:first").slideToggle();  
});  
$("#btnAppend").click(addWelcome);
```

Here in the preceding code you can see that the button click events are bound to the event handlers. The click directly binds to the click handler. Here `callSelectors` and `addWelcome` are the event handlers. You can also use the generic bind method to specify the event and the event handler.

5. In addition to the normal DOM manipulation and selectors, jQuery also supports some cool effects that you can apply. For instance, consider the following code:

```
$("p").hide(1000, function () { alert("a p are hidden"); });
$("p").show();
$("p").fadeOut(4000);
$("p").fadeIn(4000);
$("p").animate({ "fontSize": "200%" }, "slow");
```

These methods allows the DOM elements to animate. Each of these API supports you to send:

- **Seed:** This bears the value that specifies the time to complete animation
- **Callback:** A method that will be called after the animation finishes

Here in the first call, we pass all the arguments. You will notice that alerts will appear after each P is hidden. `fadeOut` and `fadeIn` are fade effects. `animate` allows you to pass styles such that the style gets animated to a certain value. Here in our case the `fontSize` value has been increased to 200 percent.

6. Finally, AJAX is one of the most important parts nowadays to bring life to a site. AJAX is asynchronous call to a server that allows you to get a response from the server without posting back the page. AJAX is used to handle partial updates on the page:

```
$.ajax({
    url: "about.aspx",
    type: "GET",
    beforeSend: function () {
        $("#ajaxResponse").html("<center><img src='images/
progressbar.gif' /></center>");
    },
    success: function (data) {
        if (data)
            $("#ajaxResponse").html(data);
    },
    error: function (req, e) {
        alert(req.status + " " + req.statusText);
    }
});
```

This code calls the URL `about.aspx` on the server with the `GET` method. `$.ajax` is the jQuery implementation of AJAX. The `beforeSend` function is called before the server request is made. It is generally used to configure the call. In our case, we have specified an image which will be loaded in the document. The success gives the output of the page. `data` being the output, we place on a `div` tag predesigned on the page. The `error` callback is called when some error has occurred on the page. For instance, it will show **404 not found** when `about.aspx` is not found.

How it works...

jQuery is a JavaScript library that runs on top of the existing JavaScript API available on the browser but also solves the following few key problems that the code running on the browser has:

- ▶ It reduces the cross-browser API problems of JavaScript
- ▶ It provides a superior API that requires minimal code to be written by the user
- ▶ It supports extension to itself

A large number of extensions or plugins for the jQuery have already been made available to the world which can be easily plugged into the source with jQuery main JavaScript library to generate a beautiful UI. jQuery also has support for HTML5 that has been introduced lately.

There's more...

jQuery is not limited to what is previously shown. Let us check some advanced topics on jQuery in this section.

Extending the jQuery library

jQuery supports extension. There are a large number of extensions to jQuery available on the market which took help from jQuery to implement their own JavaScript library. After learning jQuery, you would also feel like creating your own extension. The method that is used to extend jQuery is named as `extend`. Let us look at the following code:

```
(function($) {  
    $.fn.extend({  
        Value1 : 20,  
        myMethod : function(msg) {  
            alert(msg + "value : " + this.Value1);  
        }  
    });  
})(jQuery);
```

The preceding code will add a method called `myMethod` inside the `jQuery` library and hence you can call this method directly to get an alert with the message passed to the method. For a function, you can also define global variables. In case of this code, the `Value1` variable is defined as a global variable and hence all the methods that are defined within the extension can access `Value1`.

See also

- ▶ Refer to the following link:

<http://bit.ly/JQueryTut>

Working with task-based asynchronous `HttpHandlers` and `HttpModules`

The introduction to **async** in .NET 4.5 has opened many alternatives. ASP.NET not being an exception to this has introduced new ways of writing async operations within it. The asynchronous `HttpHandlers` and `HttpModules` classes are not new to the system, but because of task-based asynchrony available in the system, the use of asynchronous handlers and modules becomes more relevant and useful. We know tasks in async do not employ a new thread most often. It uses `SynchronizationContext` to switch between an application end thread with devices which include I/O, network, and so on. Now if you are dealing with such an operation, you cannot block a thread that has been dedicated for the entire processing. The better approach would be to release the thread that is currently running and dedicate another thread when the process finishes. The asynchronous `HttpHandlers` and `HttpModules` classes are meant for this and hence are very useful in the context of task-based async operations.

In this recipe, let us look at how exactly async handlers, and modules, work and write a custom module for our ASP.NET application.

Getting ready

Before we begin explaining asynchronous `HttpHandlers` and `HttpModules` classes, it is important to know what exactly `HttpHandlers` and `HttpModules` are. During the phase of any request processing, `HttpRuntime` passes through some phases. During these phases, the request gets processed into a response and is sent back to the client. The process of transformation is done by `HttpHandlers`. `HttpModules` include events that are associated with the runtime during the phase of processing.

Every page is a handler, and every time a page is requested, the `ProcessRequest` method of the page is called. The authentication/authorization are the modules that are invoked to validate the authenticity of the request. `HttpHandlers` and `HttpModules` together make `HttpPipeline`.

How to do it...

1. Start an ASP.NET application and create a class.
2. Create a class and call it TaskModule.
3. Inherit the class from IHttpModule and put your task-based async method.

Once you have the method defined, create a handler with (object, EventArgs) arguments such that we can post the request through EventHandlerTaskAsyncHelper:

```
private async Task PageEventHandlerAsync(object caller, EventArgs e)
{
    await GetHtmlPage("http://www.abhisheksur.com");
}
private async Task<string> GetHtmlPage(string url)
{
    using (WebClient client = new WebClient())
    {
        var result = await client.DownloadStringAsync(url);
        return result;
    }
}
public void Init(HttpApplication context)
{
    EventHandlerTaskAsyncHelper helper = new EventHandlerTaskAsyncHelper(PageEventHandlerAsync);
    context.AddOnPostAuthorizeRequestAsync(helper.BeginEventHandler, helper.EndEventHandler);
}
```

Here in the code, we have used GetHtmlPage to define the async code. We have wrapped up the method inside PageEventHandlerAsync such that we can use it to EventHandlerTaskAsyncHelper. This class makes the conversion to begin and end patterns.

4. We pass the begin and end methods to the AddOnPostAuthorizeRequestAsync module. You can use your own module as well.
5. Add a Web.config entry for the module and run. When AuthorizeRequest is called, it will automatically call the GetHtmlPage method to get a result asynchronously.
6. Create another class and now we call it TaskHandler.
7. Inherit from HttpTaskAsyncHandler. This class inherits from IAsyncHttpHandler and also performs all the important tasks to actually call a task for HttpHandler.

8. Override the `ProcessRequestAsync` method and write your content, like so:

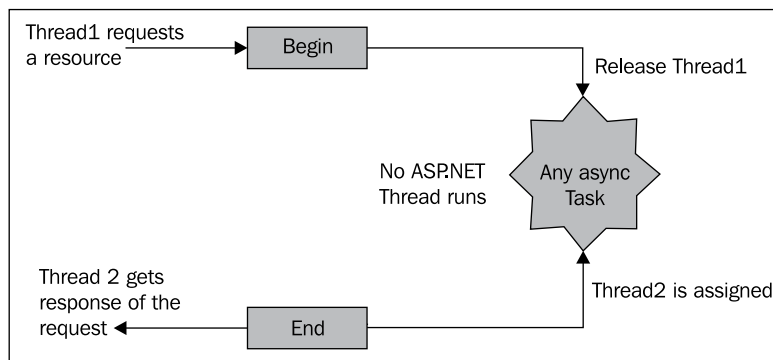
```
public override async Task ProcessRequestAsync(HttpContext
context)
{
    using (WebClient client = new WebClient())
    {
        var result = await client.DownloadStringAsync("http://www.
abhisheksur.com");
        context.Response.Write(result);
    }
}
```

In the preceding code, `WebClient` is used to get the content and write to the response.

9. Both `HttpHandler` and `HttpModule` need to be configured in `web.config`.

How it works...

In case of synchronous `HttpHandler`, the same thread is engaged in serving the request that has invoked by the request and hence, we need to block the thread even when the async operation does not need our thread to execute. Let us suppose you need to send an e-mail through the request. Doing it in a synchronous block will make the calling thread block unnecessarily for the time in it uploads the data through the network and returns back with a status message. Waiting unnecessarily like this for a long time often imposes a lot of resource cost on the server and hence, degrades the performance of the entire website. On the other hand, asynchronous handlers or modules are meant for such a scenario where the thread becomes available as soon as the control gets passed to the component and it does not wait. The calling thread returns back to the thread pool and is available to process another request. When the assigned task is complete, a new thread is assigned to service the request.



In the preceding diagram, it is depicted how the async task gets executed with asynchronous `HttpModules` or `HttpHandlers`. `Thread1` is assigned to process a resource initially. It calls the `Begin` method, requests for the resource and returns back to the thread pool. When the resource is ready through async operation, it assigns a new thread (`Thread2`), which processes the request and gets the response back.

So, from the perspective of the request-response model, it will seem that the same thread is returning the response, but in fact the threads are optimally utilized in this model and hence, reduce the downtime of the servers.

New enhancements to various Visual Studio editors

To support ASP.NET development, various Visual Studio editors are being updated. There are lot of enhancements in terms of Visual Studio that help in the development of ASP.NET applications to be easier and more productive. The new Visual Studio release has enhancements on various ASP.NET components including JavaScript editor update, ASP.NET designer update, CSS editor update, and so on. All these features directly enhance the productivity of ASP.NET applications and make development easier.

In this recipe, we are going to cover some of the interesting updates to the editor that you must know.

Getting ready

To get ready for the update, let us create a new ASP.NET project using Visual Studio 2012 IDE. Add a file in the editor and call it `Default.aspx`. Add a JavaScript file into the project and a CSS file.

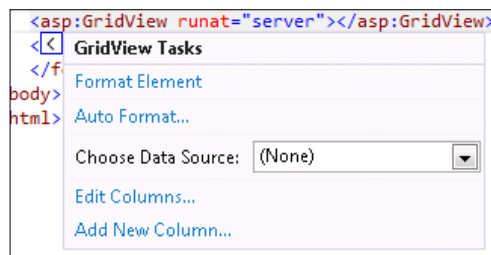
Let us look step by step at all the updates that have been made to the Visual Studio editor. We divide the Visual Studio updates into the following categories:

- ▶ HTML editor updates
- ▶ JavaScript editor updates
- ▶ CSS editor updates
- ▶ Publishing website

We are going to divide the recipe into these sections.

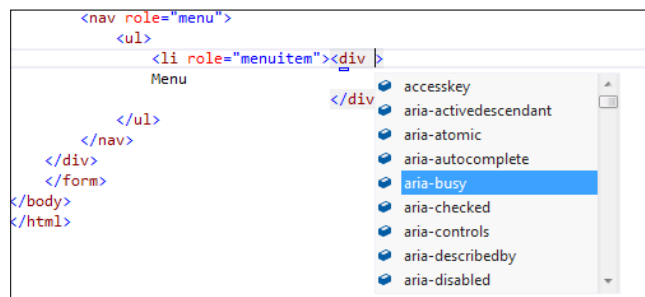
How to do it...

- ▶ HTML editor updates
 - ASP.NET is built up with lot of complex controls. For a long time, the Visual Studio design view provided a lot of dialog boxes, which helps in configuring these complex controls. But most developers do not like to go to the design view nowadays and hence, they miss these dialogs. Visual Studio 2012 IDE comes up with these dialogs in the source view. They are context-aware tasks that come into the source view on a particular control at a particular time. These features are named as **Smart Tasks** in Visual Studio:



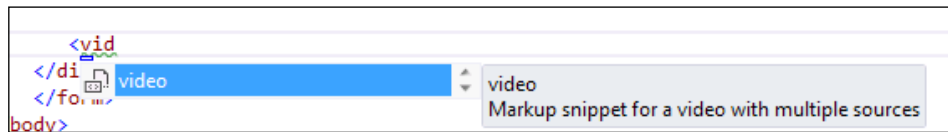
In the screenshot, GridView is declared and a SmartTask dialog appears when we press *Ctrl + .*(dot). After editing the control from this dialog, the source view gets updated with the appropriate HTML.

- Writing accessible websites is becoming important. The *aria-* attributes are important declarations on how the websites need to layout. The *aria* attributes are fully supported by Visual Studio now:



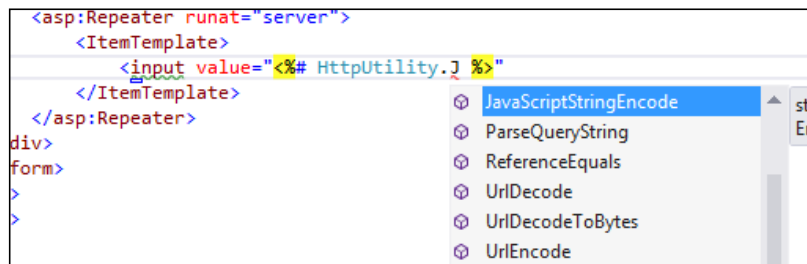
The *aria* attributes are HTML5 constructs and all the semantic of *aria* attributes are supported.

- As the world is moving towards HTML5, Visual Studio is providing snippets directly in the source view which allows you to place HTML5 constructs:



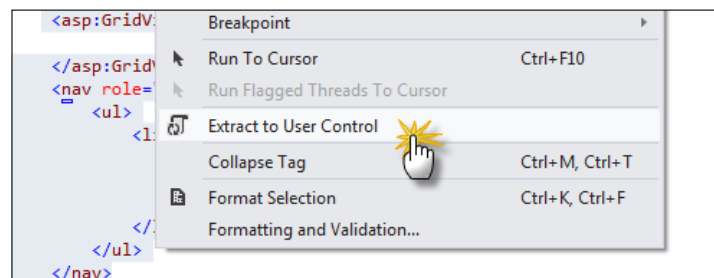
The HTML5 code snippets can be used directly inside the design view to implement HTML5 documents quickly.

- Visual Studio HTML editor now supports IntelliSense for a server-side code in HTML:



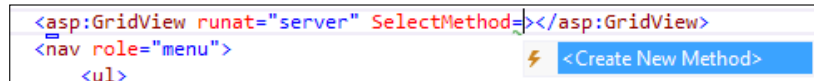
You can see, even the HTML controls are capable of providing IntelliSense inside server-side tags.

- You can extract a user control directly by selecting the HTML within a page. Generally when the page becomes more and more complex, we tend to create user controls to make a reusable component. Visual Studio provides an option to create a user control directly from the right-click menu as shown in the following screenshot:



You can see that you can right-click on the selection and extract an user control directly out of it.

- One of the best inclusions in Visual Studio HTML editor is the automatic event handler generation. Visual Studio now automatically creates method stubs while creating a design on the source view:



It automatically detects the appropriate method signature and creates it automatically for any event or method.

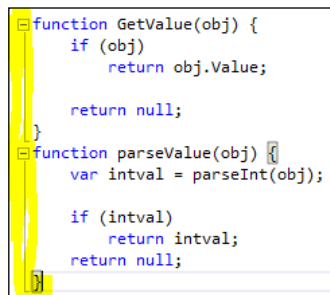
- Visual Studio HTML editor now supports smart indentation while writing the code. We often need *Ctrl + K, D* to reformat a document while writing HTML, but now Visual Studio automatically maintains this for us:



Here the indentation is maintained when we press *Enter* within any HTML element.

► JavaScript editor updates

- JavaScript editors now support code outlining, such that you can collapse some portion of the code:



You can see that the outlining is maintained within the JavaScript editor.

- JavaScript automatically matches the braces and Visual Studio shows appropriate braces that match the current brace.
- JavaScript editors supports **Go to Definition**. So when we right-click on a JavaScript call and select **Go To Definition**, the cursor moves to the actual definition of the method.

- A VS document is also supported by JavaScript editors. A signature element can be used to specify the JavaScript documentation. Even signature overloads are also supported by the IDE:

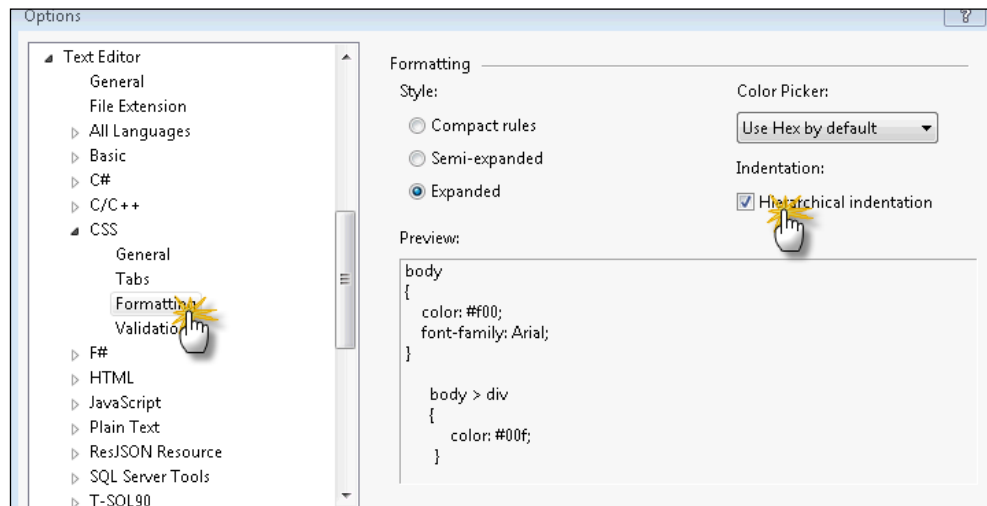
```
function parseValue(obj, defaultval) {
    ///<signature>
    ///    <summary>Parses the integer value from a
string</string>
    ///    <param name="obj" type="String">Provides string
equivalent of an integer</param>
    ///    <returns type="int"/>
    ///</signature>
    ///<signature>
    ///    <summary>Parses the integer value from a
string</string>
    ///    <param name="obj" type="String">Provides string
equivalent of an integer</param>
    ///    <param name="defaultValue" type="int">Provides
an integer that is passed by default</param>
    ///    <returns type="int"/>
    ///</signature>
    var intval = parseInt(obj);

    if (intval)
        return intval;
    return defaultval;
}
```

Here the code offers two document for its two overloads and when the method is called, the two overloads are shown to the user.

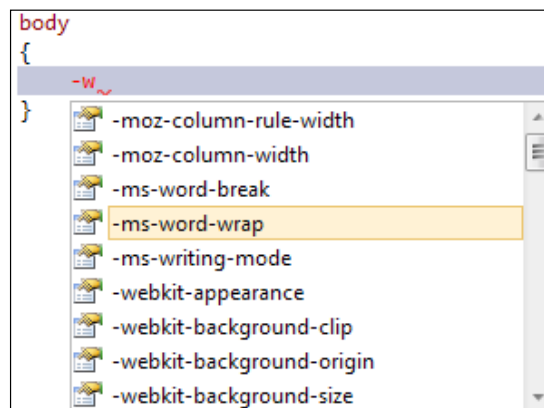
- JavaScript files automatically maintain a central list of files for referencing external files. For instance, if you have added a jQuery file directly into the central repository of the project, IntelliSense automatically detects the reference and shows the IntelliSense menu for all the methods from jQuery.
- CSS Editor updates
 - The IntelliSense menu for the CSS editor now supports title case filters or even filters based on CSS properties. For instance, if you start typing "border", it will list the borders only in the IntelliSense menu.

- ❑ CSS editor supports hierarchical rules such that the child CSS rules are indented:



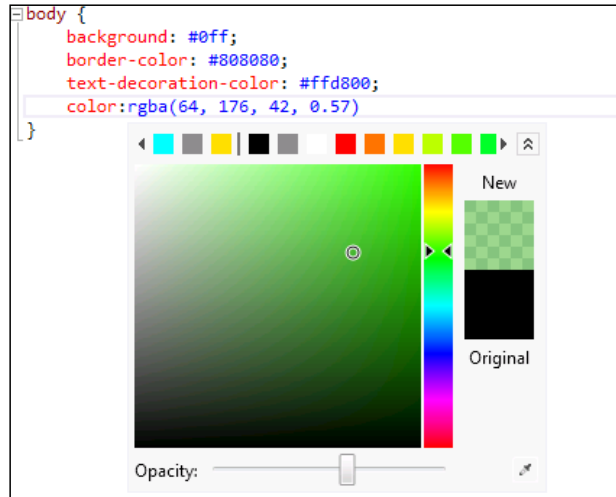
In the preceding screenshot you can see when hierarchical indentation is selected; the `div` tag inside the `body` element will be indented as a child automatically.

- ❑ CSS editors now supports a large number of vendor-specific CSS. All of them are listed in the IntelliSense menu of the Visual Studio CSS editors:



Here you can see the `-ms-word` CSS specific to Microsoft Word, while `-webkit` belongs to Chrome and `-moz` belongs to Mozilla.

- ❑ Visual Studio CSS editor has been enhanced with a lot of controls. For instance, the CSS editor has an in-built color picker within the IDE that allows you to pick up a color while defining the same within the CSS.



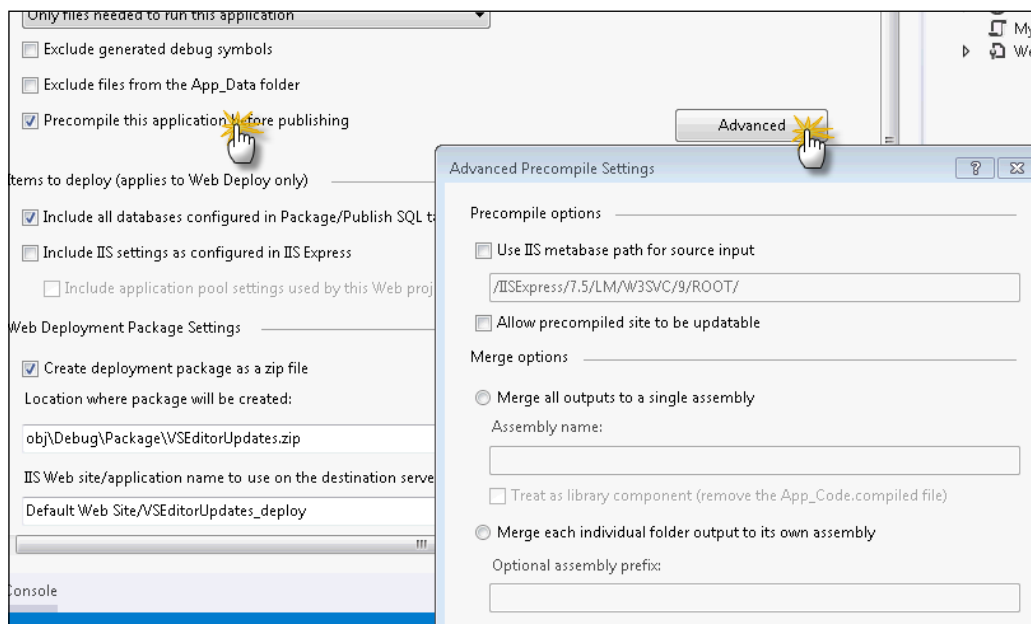
Here in this screenshot, you can see the color picker has been opened to define the specific color for the `body` element.

- ❑ CSS editor now supports the creation of custom regions such that the region can be collapsed using outlines. The open region matches the respective end region to provide an outline to the whole code within the scope:



Here you can see the region is defined for the `body` element which can be expanded and collapsed into an outline. This helps in reducing complexity in code.

- ❑ The **Page Inspector** tool is a new addition to Visual Studio that allows you to examine the actual output of a section of code with the corresponding HTML markup without opening the HTML code in the IDE. The **Page Inspector** tool helps in determining the actual output of the end product and quickly examines the output form within the Visual Studio IDE.
- Publishing enhancements
 - ❑ Visual Studio web applications support profiles. You can maintain a number of publish profiles within the project and depending on the profile that has been selected, the project gets published. The profiling option has been added to MSBuild and works directly while the project is built.
 - ❑ ASP.NET projects also give an option to precompile the application before the publishing option under **Package/Publishing Web Properties** page. This option lets the user to merge the site's content while publishing or packaging the project:



Here you can see the **Package/Publish Web Properties** page provides an option to precompile application before publishing. You can select the **Advanced** settings to choose how the application will merge assemblies.

There's more...

Visual Studio has been enhanced to support modern trends of development. Let us look into some of the other enhancements that have been made to the IDE to support current trends of development.

Configuration changes in ASP.NET 4.5

Visual Studio configuration plays a vital part in any website. IIS 7 introduced managed web hosting, which can read `web.config` of a certain website and detect the configurations that has been set to the website. Hence the IIS can be configured directly from Visual Studio itself by writing correct entries in the `web.config` file.

We generally define the application settings, `connectionStrings`, `httpHandlers`, modules, and so on, inside a web configuration file but there are certain configurations that actually determine the overall architecture of the website. ASP.NET 4.5 introduces such architectural configuration sections that can configure the architecture of the IIS where it is going to be hosted. Let us define some of the interesting changes to the configuration file:

- ▶ `<httpRuntime>` introduces a new `encoderType` attribute in configuration that enables you to set the default encoder for the site. The default defined in the template is `AntiXSS`.
- ▶ The `requestValidationMode` attribute of `httpRuntime` defines how the request is validated to the server before calling `HttpPipeline`. The default settings for any web application in .NET 4.5 are set to 4.5, which means deferred validation.
- ▶ The default value of `runAllManagedModulesForAllRequests` in the `modules` section of `<system.webServer>` is set to `false`. When this setting is set to `true`, the IIS 7 ASP.NET routing is automatically configured. All the requests made to the website directly calls the modules in the website of IIS 7 such that if routing is configured on the server, it calls the appropriate module for routing automatically.

These small changes to the configuration have been made to the ASP.NET template such that these changes are readily available to any site that is created for the first time and not configured.

See also

- ▶ Refer to the following link:
<http://bit.ly/ASP45VS>

5

Enhancements to WPF

The goal of this chapter is to introduce the enhancements that have been made to WPF 4.5 applications. After reading the chapter, you will have an idea of how to build a WPF application and also know the latest updates to WPF technology. In this chapter we are going to cover the following recipes:

- ▶ Getting started with WPF and its major enhancements in .NET 4.5
- ▶ Building applications using MVVM pattern supported by WPF
- ▶ Using the Ribbon User Interface in WPF
- ▶ Using WeakEvent pattern in WPF

Introduction

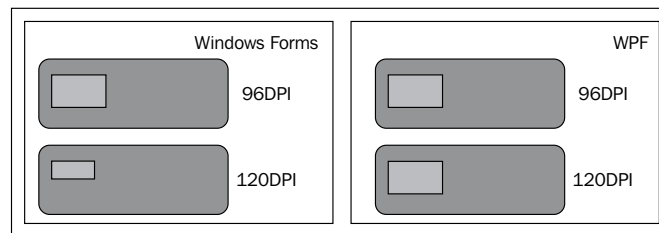
With the introduction to modern day styles of application development, people are more and more inclined towards UI technologies. UI became the primary concern for customers. Technologies, such as HTML5, CSS3, and DirectX utter the same voices. Customers are also nowadays more inclined towards presentation with respect to look and feel rather than functionalities.

WPF comes as a new technology from Microsoft that deals with these problems. It has lots of advantages. Let me introduce a few of its features.

Device Independent Pixel (DPI)

WPF introduces **Device Independent Pixel (DPI)** settings for the applications built with it. For a window, it is very important to calculate how many dots per inch (DPI) the screen can draw. This is generally dependent on the hardware device and operating system in which the application runs and also how the DPI settings are applied on the device. Any user can easily customize these settings and hence, make the application look horrible. Windows forms application uses a pixel-based approach. So with changing DPI settings, each control will change its size and look.

WPF addresses this issue and makes it independent of DPI settings of the computer. Let's look at how it is possible.



Let's say you have drawn a box, just like the one in the figure, which is one inch long in the **96DPI** screen. Now if you see the same application in **120DPI** settings, the box will appear smaller. This is because the things that we see on the screen are totally dependent on DPI settings.

In the case of WPF, this is modified to a density-based approach. That means when the density of a pixel is modified, the elements will adjust them accordingly and hence, the pixel of WPF application is Device Independent Pixel. As you can see in the figure, the size of the control remains the same in the case of the **WPF** application, and it takes more pixels in the case of the **120DPI** application to adjust the size properly.

Built-in support for graphics and animation

As WPF applications are being rendered within a DirectX environment, it has major support for graphics and animation capabilities. A separate set of classes are present that specifically deal with animation effects and graphics. The graphics that you draw over the screen are also vector-based and are object-oriented. That means, when you draw a rectangle in a WPF application, you can easily remove it from the screen, as the rectangle is actually an object which you always have a hold on. In a traditional Windows-based application, once you draw a rectangle, you can't select it individually. Thus programming approach in the case of WPF is completely different and more sophisticated than a traditional graphics approach. We will discuss graphics and animation in more detail in a later section of the article.

Redefine styles and control template

In addition to graphics and animation capabilities, WPF also comes with huge flexibility to define the styles and control template. A style-based technique, such as you might come across with CSS is a set of definitions which defines how the controls will look like when it is rendered on the screen. In the case of traditional Windows applications, styles are tightly coupled with each control, so that you need to define color, style, and so on for each individual control to make it look different. In case of WPF, styles are completely separated from the UIElement. Once you define a style, you can change the look and feel of any control by just putting the style on the element.

Most of the **UIElement** properties that we generally deal with are actually made using more than one individual element. WPF introduces a new concept of templates, which you might use to redefine the whole control itself. Say for instance, you have a **CheckBox**, which has a rectangle in it and a **ContentPresenter** (one where the caption of the **TextBox** appears). Thus you can redefine your checkbox and put a **ToggleButton** inside it, so that the check will appear on the **ToggleButton** rather than on the rectangle. This is very interesting. We will look into more detail on Styles and **ControlTemplate** later.

Resource-based approach for every control

Another important feature of WPF is resource-based approach. In the case of traditional Windows applications, defining styles is very hectic. So if you have 1,000 buttons, and you want to apply the color gold to each button, you need to create 1,000 objects of color and assign each to individual elements. Thus, it makes it very resource hungry.

In WPF, you can store styles, controls, animations, and even objects as a resource. Thus, each resource will be declared once when the form loads itself, and you may associate them to the controls. You can maintain a full hierarchy of styles in a separate file called `ResourceDictionary`, from which styles for the whole application will be applied. Thus the WPF application could be themed very easily. Moreover, to deal with styles, Microsoft has introduced a new tool called **Expression Blend** which is more suited for the designers and allows generating styles and resources directly after working with the designer toolsets. It is very easy and necessary when dealing with complex designs and textures. Visual Studio 2012 Professional and above, automatically ships with Expression Blend which can be of great use while developing cool styles for an application. The Expression Blend tool comes with some predefined styles which also add up to its functionalities.

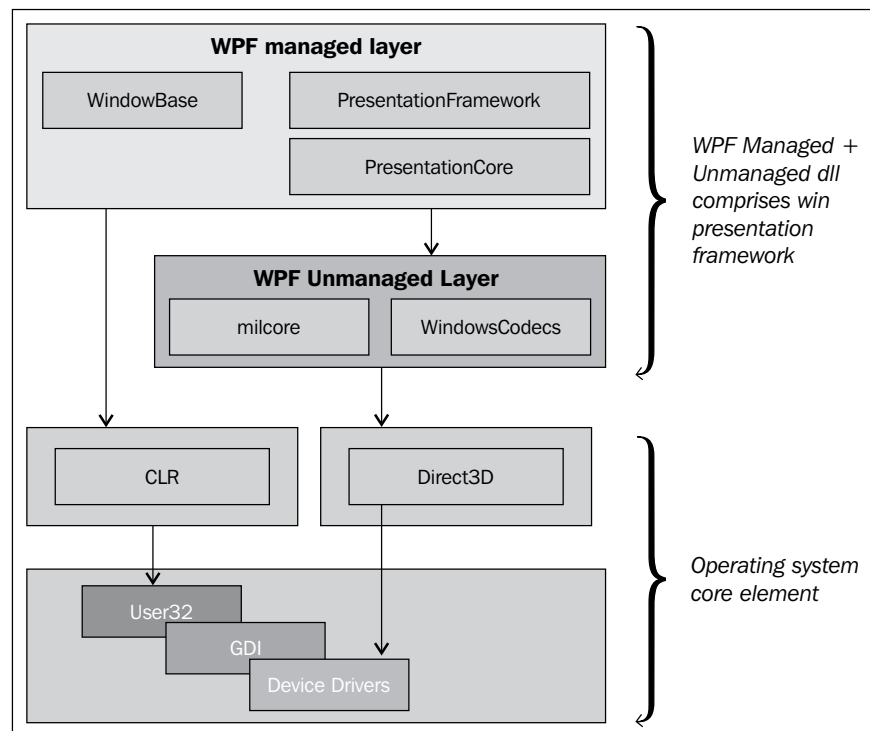
New property system and binding capabilities

Here, I must introduce the new property system introduced with WPF. Every element of WPF defines a large number of dependency properties. The dependency properties have stronger capabilities than the normal properties. Thus when I define our new property, we can easily register our own property to any object we wish to. It will add up to the same observer that is associated to every object. As every element is derived from `DependencyObject` in its object hierarchy, each of them contains `DependencyObserver`. Once you register a variable as Dependency property, it will create a room on the observer associated with that control and set the value there. We will discuss this in more detail in later sections of the series.

In this chapter we are going to discuss a lot of enhancements to the presentation layer and introduce WPF to get you started.

Getting started with WPF and its major enhancements in .NET 4.5

WPF as of now has gone through a lot of changes to its core and made it richer. The idea behind the WPF system is to make sure that the object model can take advantage of all the hardware and technological advancements that are going on in other areas of .NET, such as asynchronous programming, hardware acceleration, animation, data binding, and so on.



For every new technology, it is very essential to have a clear idea about its architecture. So before beginning your application, you must grab a few concepts. If you would not like to know WPF in detail, please skip this recipe. As mentioned earlier, WPF is actually a set of assemblies that build up the entire framework. These assemblies can be categorized as follows:

- ▶ **Managed Layer:** Managed layer in WPF is built using a number of assemblies. These assemblies build up the WPF framework, and communicate with the lower level unmanaged API to render its content. The few assemblies that comprise the WPF framework are:
 - ❑ `PresentationFramework.dll`: This creates the top level elements, such as layout panels, controls, windows, styles, and so on.

- ❑ `PresentationCore.dll`: This holds base types, such as `UIElement`, visual from which all shapes and controls are derived in `PresentationFramework.dll`.
- ❑ `WindowsBase.dll`: They hold even more basic elements which are capable of being used outside the WPF environment, such as dispatcher object, and dependency objects. I will discuss each of them later.
- ▶ **Unmanaged Layer** (`milcore.dll`): The unmanaged layer in WPF is called **milcore** or **Media Integration Library Core**. It basically translates the WPF's higher-level objects, such as layout panels, buttons, animation, and so on into textures that `Direct3D` expects. It is the main rendering engine of WPF.
- ▶ `WindowsCodecs.dll`: This is another low-level API which is used for imaging support in WPF applications. `WindowsCodecs.dll` comprises a number of codecs which encode/decode images into vector graphics that would be rendered into the WPF screen.
- ▶ **Direct3D**: This is a low-level API in which the graphics of WPF are rendered.
- ▶ **User32**: This is the primary core API which every program uses. It actually manages memory and process separation.
- ▶ **GDI and Device Drivers**: GDI and Device Drivers are specific to the operating systems which are also used from the application to access low-level APIs.

This is the basic architecture of the WPF application. Now let us take a look at the major enhancements that have been made to WPF recently:

- ▶ Support for Async programming style
- ▶ Easier ways to access collection from non-UI threads
- ▶ Styles, triggers, and animation to objects
- ▶ Markup extensions

How to do it...

In this recipe, we are going to cover these concepts such that at the end of this recipe, you will have the basic idea about these concepts.

The following steps will help you to learn the major enhancements of WPF.

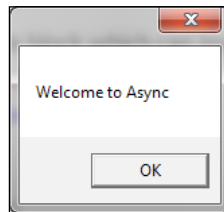
1. Start a new WPF application. Once the application gets loaded, you will see `MainWindow.xaml`.
2. Add a button to the XAML code and put a click handler to it. We call the button as `Click Me`.

```
<Grid>
    <Button Content="Click Me" Click="btnClickMe_Click" />
</Grid>
```

3. When the button is clicked, we call an `async` code block which can invoke an `async` operation. For simplicity we put an awaitable delay.

```
private async void btnClickMe_Click(object sender, RoutedEventArgs e)
{
    await Task.Delay(5000);
    MessageBox.Show("Welcome to Async");
}
```

This code places a message box just after five seconds:



4. When the WPF application starts, it actually creates two threads automatically. One is the rendering thread, which is hidden from the programmer and the other one is the dispatcher thread, which ties all the UI elements created together. Just like in other Windows environments, any control created on WPF will have thread affinity towards the dispatcher thread. In WPF every UI element is derived from `DispatcherObject`, and thus the reference of the dispatcher thread is inherent within every object. We can also make use of `Dispatcher` to call `async` code blocks. For instance the `Dispatcher` class has methods, such as `InvokeAsync` which runs an `async` block and returns an awaitable.

```
public async void CallingMethod()
{
    await Dispatcher.CurrentDispatcher.InvokeAsync(mymethod);
}
```

In the preceding code the `mymethod` is called through `Dispatcher`. The `await` can be applied to this call when the `InvokeAsync` method is used. The `InvokeAsync` method returns a `DispatcherOperation` object. So if `mymethod` returns some object, you can have the returned value in this object. The internally used task object is also wrapped inside this object.

```
Func<string> method = this.myMethod;
DispatcherOperation<string> dispatcherOperation = Dispatcher.
CurrentDispatcher.InvokeAsync(method);
dispatcherOperation.Task.Wait();
MessageBox.Show(dispatcherOperation.Result);
```

When the string is returned from the method, it is shown to the user in `MessageBox`. The `InvokeAsync` method, like any async block, also supports `CancellationToken`.

5. The WPF controls have thread affinity. That means the objects created from one thread cannot be accessed by another thread. We generally call the dispatcher thread to handle collection modifications. As `Dispatcher` points to the UI thread, the call which needs to access collection objects will work.

But calling the dispatcher thread several times will make the code very ugly. Using `EnableCollectionSynchronization` on `BindingOperations` will allow the collection to automatically handle `ThreadAffinity`. For instance:

```
//Creates the lock object
private static object _lock = new object();
//Enable the cross access to this collection
BindingOperations.EnableCollectionSynchronization(_persons, _
lock);
```

Now if you add an object to the list in another thread, it will automatically handle that.

6. WPF allows you to specify custom types to be associated with its properties. These custom properties are called **Markup Extensions**. The markup is annotated with markup extensions, such as `StaticResource`, `DynamicResource`, `Binding`, and so on. You can also define your own markup extension. Let us create a new style for our button and place it inside resources.

```
<Window.Resources>
    <Style x:Key="btnStyle" TargetType="{x:Type Button}">
        <Setter Property="Background" Value="Bisque" />
        <Setter Property="Foreground" Value="Red" />
    </Style>
</Window.Resources>
```

In the preceding code we define a style that can be applied to `Button`. The `Style` property is a special type that holds a collection of `Setters` which allows you to find the properties of a control and set its values such that it can be applied as a whole to the instance of that control. It is the same as CSS works in the web.

7. To apply this style we use `Style="{StaticResource btnStyle}"`. `StaticResource` is a custom type defined in the WPF system which is known as Markup Extension.
8. Create a new class called `ReflectionExtension` and extend the same from `MarkupExtension`. In every `MarkupExtension`, we need to override `ProvideValue` to get the result. In our case we get the names of all the methods, properties, events, and so on for a type passed into it.

```
public class ReflectionExtension : MarkupExtension
{
```



```
public Type CurrentType { get; set; }
public bool IncludeMethods { get; set; }
public bool IncludeFields { get; set; }
public bool IncludeEvents { get; set; }

public ReflectionExtension(Type currentType)
{
    this.CurrentType = currentType;
}

public override object ProvideValue(IServiceProvider
serviceProvider)
{
    if (this.CurrentType == null)
        throw new ArgumentException("Type argument is not
specified");

    ObservableCollection<string> collection = new
ObservableCollection<string>();
    foreach(PropertyInfo p in this.CurrentType.
GetProperties())
        collection.Add(string.Format("Property : {0}",
p.Name));

    if(this.IncludeMethods)
        foreach(MethodInfo m in this.CurrentType.GetMethods())
            collection.Add(string.Format("Method : {0} with
{1} argument(s)", m.Name, m.GetParameters().Count()));
    if(this.IncludeFields)
        foreach(FieldInfo f in this.CurrentType.GetFields())
            collection.Add(string.Format("Field : {0}",
f.Name));
    if(this.IncludeEvents)
        foreach(EventInfo e in this.CurrentType.GetEvents())
            collection.Add(string.Format("Events : {0}",
e.Name));

    return collection;
}
}
```

9. Add a namespace to the header to point to the markup extension. We add `xmlns:local="clr-namespace:WPFFirstAppSample"` in the header window tag of the window as an attribute and add a `ListBox` to the `StackPanel`.

```
<ListBox ItemsSource="{local:Reflection {x:Type Grid},
    IncludeMethods=true, IncludeFields=true,
    IncludeEvents=true}"
    MaxHeight="200" />
```

The `ListBox` property calls the `ProvideValue` with the type passed in (`Grid` in our case) and enumerates all the members, fields, events, and so on for it. `MarkupExtension` for events is also supported by the latest release.

10. `VirtualizingStackPanel` is a special container which allows paging more elegantly. For `ListBox` if you specify virtualization, the `ListBoxItem` property will not be created initially, rather will be called on the fly when the data is scrolled. Let's add `VirtualizingStackPanel` on `ListBox` to gain performance.

```
<ListBox ItemsSource="{local:Reflection {x:Type Grid},
    IncludeMethods=true, IncludeFields=true, IncludeEvents=true}"
    MaxHeight="200"
        VirtualizingStackPanel.IsVirtualizing="True"
        VirtualizingStackPanel.
VirtualizationMode="Recycling"
        VirtualizingStackPanel.ScrollUnit="Pixel"
        VirtualizingStackPanel.CacheLength="2,3"
        VirtualizingStackPanel.CacheLengthUnit="Page"
/>
```

Here in the code `VirtualizingStackPanel.IsVirtualizing` states that the container needs to be virtualized. The `Mode` is `Recycling`, which indicates that the same `ListBoxItem` will be recycled for the scrolling. `ScrollUnit` is `Pixel`, which means the scrolling will be based on each pixel. You can also specify `CacheLength` and `CacheLengthUnit`. Each of them defines how many items need to be cached for virtualization. For heavy list item, it is better to have a page cached. This will make the scrolling smoother.

1. Now let's add a `Rectangle` to the XAML. We use `EventTrigger` to trigger an animation to the `Rectangle`.

```
<Rectangle Width="50" Height="50">
    <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Loaded">
            <BeginStoryboard>
                <Storyboard RepeatBehavior="Forever">
                    <DoubleAnimation Storyboard.
TargetProperty="Width"
```

```
                From="50" To="0" AutoReverse="True"
Duration="0:0:5" ></DoubleAnimation>
                <DoubleAnimation Storyboard.
TargetProperty="Height"
                From="50" To="0" AutoReverse="True"
Duration="0:0:5"></DoubleAnimation>
                <ColorAnimation From="Blue" To="Green"
Duration="0:0:5"
                Storyboard.
TargetProperty="Fill.Color" AutoReverse="true"/>
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Rectangle.Triggers>
</Rectangle>
```

In the preceding code `EventTrigger` is used to hook on the `Loaded` event of `Rectangle`. The initial color of `Rectangle` is made blue. `Storyboard` is a collection of animation. Here, `DoubleAnimation` is used to animate the width and Height values (which are of type double), and `ColorAnimation` is used to animate the Fill color of the `Rectangle`.

How it works...

Among so many changes to the WPF system recently, the compiler team has made the right adjustments to the WPF code to orchestrate the asynchronous calls to the members from the XAML. Any event from XAML can call the `async` method and code, as during compilation the binding is automatically done to the original method with the BAML (Binary XAML).

XAML is a replacement of C# code. It allows you to create objects inside it, specifying other objects. Markup extensions are special classes that can be used inside curly braces, such that when the XAML is parsed, it automatically detects these markup extensions and creates respective calls to the object. Each `MarkupExtension` class has a `provideValue` implementation. When the object is parsed, the `ProvideValue` implementation is called to get the actual type specified. For instance:

```
<Button Foreground="{StaticResource fground}" />
```

In the preceding code `Foreground` is specified with a markup extension. The `StaticResourceExtension` is a class for which the `ProvideValue` implementation automatically tries to parse the resources and find the brush with the `Foreground` key. When the object is found it is assigned to the `Foreground` property. You can specify properties for a markup extension as well which will be separated using a comma.

Animation of WPF has been simplified pretty much. Animation can be defined as the changes in values over time. You can specify `Storyboard` to define the timeline of the entire animation. The idea is to define animation for a property. For instance, say you want to animate the value of an integer variable, you can use `Int32Animation`. Similarly, when you want to animate value of a double type, you can use a `DoubleAnimation`, or when you want to animate a color, you can use `ColorAnimation`. Each of these animations have been well defined in the system, and based on the type of the property you can specify the appropriate animation over time.

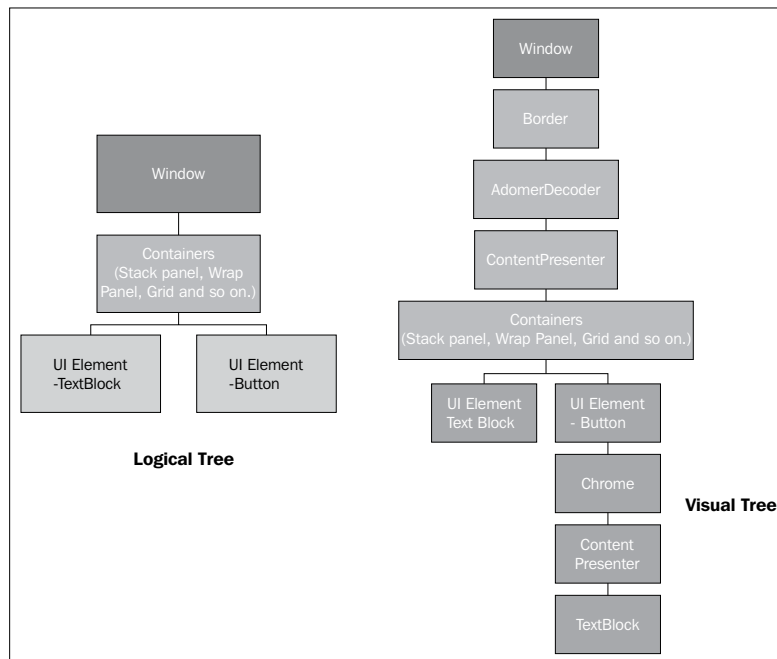
There's more...

The WPF is pretty new to the developers even though it has gone through some of the major enhancements already and is in version 4.5. There are a few concepts that you need to understand to know it better. Let us discuss them here.

What are Visual Trees and Logical Trees?

Every programming style contains some sort of `LogicalTree` which comprises the overall program. `LogicalTree` comprises the elements as they are listed in XAML. Thus, they will only include the controls that you have declared in your XAML.

`VisualTree` on the other hand, comprises the parts that make up the individual controls. You do not generally need to deal with `VisualTree` directly, but you should know how each control is comprised, so that it will be easier to build custom templates using this.



What are RoutedEvents?

`RoutedEvents` are very new to the C# language, but for those who are coming from JavaScript/web tech, you would have found it in your browser. Actually there are two types of `RoutedEvents`. One which bubbles through the Visual Tree elements and another which tunnels through the Visual Tree elements. There is also `Direct RoutedEvent` which does not Bubble or Tunnel.

When `RoutedEvent`, which is registered, is invoked, it Bubbles/Tunnels through the Visual Tree elements and calls all the registered `RoutedEventHandlers` associated within the Visual Tree one by one.

To discriminate between the two, WPF demarcated events with *Preview**** as the events which are Tunneled, and just ***** for the events that are Bubbled. For instance, `IsPreviewMouseDown` is the event that tunnels through the Visual Child elements while `MouseDown` Bubbles. Thus Mouse Down of the Outermost element is called first in case of `IsPreviewMouseDown` while Mouse Down for the innermost element will be called first in case of `MouseDown` event.

What is DependencyObject?

Every WPF control is derived from `DependencyObject`. `DependencyObject` is a class that supports `DependencyProperty`, a property system that is newly built in WPF. Every object is derived from `DependencyObject` and hence it can associate itself in various in-built features of WPF, such as `EventTriggers`, `PropertyBindings`, `Animations`, and so on.

Every `DependencyObject` actually has an observer or a list, and declares three methods, namely `ClearValue`, `SetValue`, and `GetValue` which are used to add/edit/remove those properties. Thus the `DependencyProperty` will only create itself when you use `SetValue` to store something. Thus, it is resource saving as well. We will look at `DependencyProperty` in detail in other articles in the series.

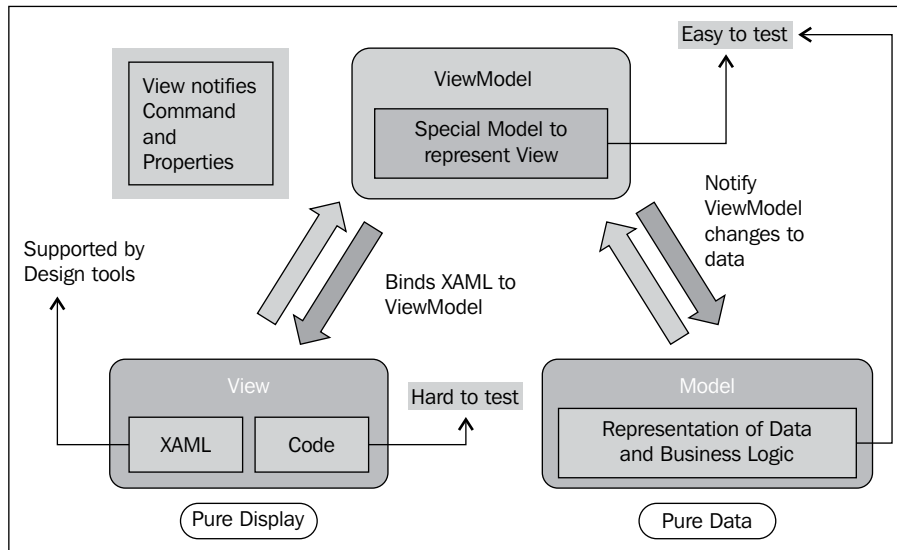
Building applications using MVVM pattern supported by WPF

WPF introduces a pattern called MVVM pattern and inherently supports it from its core. The applications built on this pattern support all the core entities of any presentable framework such that we can reuse the presentation logic in more than one application.

The **Model-View-ViewModel (MVVM)** pattern splits the user interface into three conceptual parts:

- ▶ **Model:** This represents a set of classes which points to where the data is coming from.
- ▶ **View:** This represents the visual representation of the data as UI element to which the user interacts.

- **ViewModel:** It serves as a glue between Model and View by wrapping the data coming from Model and transforming it to a user-friendly manner which can be directly presented to the View. ViewModel also controls the interactions by the user in the View with the rest of the application.

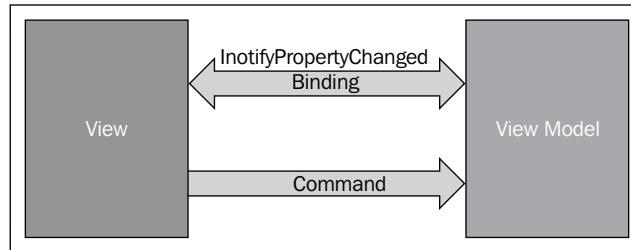


The main key components demonstrated in the preceding architecture are Model, View, and ViewModel. The ViewModel refers to Model and breaks the Model into viewable data and commands. View refers to the ViewModel to get the entire data and notifies the ViewModel due to interactions by the user. You should notice that View can also hold some code, but this code is hard to test without the actual UI being present and hence is recommended to minimize. According to the MVVM pattern, we minimize or reject any code to be placed directly into View, but rather interact with the ViewModel directly to nurture any user interaction. The ViewModel receives these interactions through Binding between UI elements with the ViewModel and notifies the Model representing data from services or database.

The MVVM pattern employs few WPF features to support it inherently. Expression Blend is an application built on MVVM model and represents the ideal example of an MVVM application. The features that employs the MVVM model from WPF are:

- **Binding:** Binding communicates through the `ModelObject` and `ViewModel` properties
- **Data Templates:** This transforms the data from ViewModel to a visual

- **Command:** This notifies the interaction from View to ViewModel



View communicates with ViewModel using Command or Binding. Binding works on data elements. ViewModel exposes the data elements which are used by the WPF controls to perform Binding. Binding in WPF hooks into the `INotifyPropertyChanged` event for the property it is bound to, and notifies any changes to the property directly.

In this recipe, we are going to take a brief tour of creating an application using MVVM model.

Getting ready

Before we actually use the MVVM model, we should at least know about two interfaces which need to be used pretty much while developing. As we have already shown that the main communication between ViewModel and View is done using Binding and Command, we need to implement two interfaces that are actually used by the WPF controls to invoke these notifications:

- `INotifyPropertyChanged`: This is an interface that raises property notifications, such that when the ViewModel raises the notification to the property, View updates the data from the ViewModel and vice versa.
- `ICommand`: It represents an individual command in which the control in the View sends direct notification to the ViewModel.

The controls that supports Command, expects an object of `ICommand` to invoke the command directly from the control. Let us implement the `ICommand` interface first. We start by creating a WPF application project and add a class library to the project. We create a class called `RegisterCommand` and start writing the code.

```
public class RegisterCommand : ICommand
{
    private Action<object> executeMethod;
    private Func<object, bool> validateMethod;
    public RegisterCommand(Action<object> executeMethod, Func<object,
bool> validateMethod)
    {
```

```

        this.executeMethod = executeMethod;
        this.validateMethod = validateMethod;
    }
    public RegisterCommand(Action<object> executeMethod)
        : this(executeMethod, e => true)
    {
    }

    public bool CanExecute(object parameter)
    {
        try
        {
            return validateMethod(parameter);
        }
        catch { return false; }
    }

    public void Execute(object parameter)
    {
        this.executeMethod(parameter);
    }

    public event EventHandler CanExecuteChanged;
    private void OnCanExecuteChanged()
    {
        if (this.CanExecuteChanged != null)
            this.CanExecuteChanged(this, EventArgs.Empty);
    }
}

```

The `ICommand` interface specifies two methods and an event:

- ▶ **CanExecute:** We need to call a delegate to ensure that the command is validated to execute. Generally when `CanExecute` returns false, the Command remains disabled in the UI.
- ▶ **Execute:** Calls the method that needs to execute when the command is performed by the View.
- ▶ **CanExecuteChanged:** The event is raised to re-evaluate the `CanExecute` for the command.

The implementation of `ICommand` can be bound to `Command` in the UI element and it invokes `CanExecute`, `Execute`, and `CanExecuteChanged` automatically when certain object state changed.

Another most important consideration of the MVVM model is the implementation of the `INotifyPropertyChanged` interface. This interface is needed by the property which is bound from the UI. `UIElement` exposes dependency properties which require property notifications to ensure that View can notify the model and vice versa. When we raise the event `PropertyChanged` for the property inside the `ViewModel`, the UI gets refreshed, and when the user interacts with the control in UI, the property gets reset automatically. We implement the interface into a class from which we inherit the `ViewModel`.

```
public class PropertyBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public void RaisePropertyChanged<T>(string propertyName, T oldVal,
    T newVal)
    {
        //We ensure that the oldValue is not equal to newValue such
        that we dont
        //raise unnecessary propertychanged notifications
        if (oldVal != null && !oldVal.Equals(newVal) && this.
        PropertyChanged != null)
            this.PropertyChanged(this, new PropertyChangedEventArgs(p
        ropertyName));
    }
}
```

Here the `INotifyPropertyChanged` interface is implemented. The interface only defines an event `PropertyChanged`. Now when we need a property to invoke change notification, we call `RaisePropertyChanged` with `oldValue` of the property and `newValue` of the property such that we ensure that event doesn't get raised unnecessarily.

How to do it...

To demonstrate how MVVM pattern works, let us take an example. We are going to create a sample application that keeps track of data inside a list and allows to edit and manipulate data for that list.

1. We start by creating the Model for the `MainWindow`. Generally we specify the `MainWindow` model directly to the opening window and maintain all other child windows from `MainWindow`.
2. `MainModel` holds references the other models (`AllUsersModel` and `UserModel`). Generally the `MainModel` creates object of all other Model and opens a `ChildWindow` from `MainWindow`.
3. We create two buttons on the `MainWindow`, called `Manage Users` and a `Add a user`. The click handler has been handled to set `DataContext` of each window that is created by `MainWindow`.

```
<Button Content="Manage Users" Click="btnManageUser_Click"/>
<Button Content="Add a User" Click="btnAddUser_Click"/>
```

```

private void btnManageUser_Click(object sender, RoutedEventArgs e)
{
    listUsers luser = new listUsers();
    luser.DataContext = this.model.ListUsers;
    luser.Show();
    this.CurrentWindow = luser;
}
private void btnAddUser_Click(object sender, RoutedEventArgs e)
{
    editUser euser = new editUser();
    euser.DataContext = this.model.NewUser;
    euser.Show();
    this.CurrentWindow = euser;
}

```

4. Here you can see the windows `lstUsers` and `editUsers` are created, and the respective `DataContext` is set. As we created the windows from `MainWindow`, it is important to note that only `Mainwindow` can close it. So you should always pass `MainModel` inside the child models and from `ChildModel` pass data to `MainModel` to help it do these tasks.
5. To close a window, we cannot do that from `ViewModel` directly, we need to do it from the View itself. To deal with this, we subscribe to the `PropertyChanged` event of `MainModel`, and we store `CurrentWindow` in a property. Hence, from the `ChildWindow` we can reset a property to close the window.
6. We define a `ListBox` property to ensure we show all the users that are currently present in the Collection. The `ListBox` is bound to the `Users` property which points to the `ObservableCollection`. Hence when the `ListUser` window is opened, it shows all the users in a list.

```

<ListBox ItemsSource="{Binding Users}" SelectedItem="{Binding
CurrentUser}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding UserName}" Margin="10"
/>
                <TextBlock Text="{Binding FirstName}"
Margin="10"/>
                <TextBlock Text="{Binding LastName}"
Margin="10"/>
                <Button Command="{Binding Delete}"
Content="Delete" Margin="10"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>

```

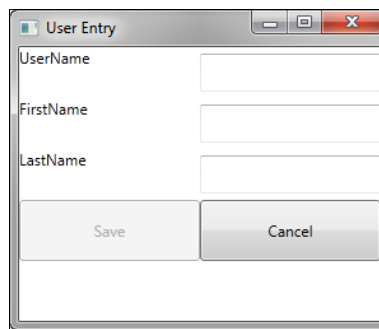
The model that is associated with the window exposes the list of users into a collection of users. The `CurrentItem` is automatically set using the `CurrentUser` property. The `ItemTemplate` for the `ListBox` defines the visual of the individual user entity:

```
public ObservableCollection<UserModel> Users { get; set; }

private UserModel _currentUser;
public UserModel CurrentUser
{
    get
    {
        return this._currentUser;
    }
    set
    {
        var cuser = this._currentUser;
        this._currentUser = value;
        this.RaisePropertyChanged<UserModel>("CurrentUser", cuser,
value);
    }
}
```

The preceding code maintains a collection of `UserModel` and a property called `CurrentUser`. When an item is selected by the user in the UI, the `CurrentUser` property is set and it raises the `PropertyChanged` notification.

7. The **User Entry** window creates an user entity. We create the **FirstName**, **LastName**, and **UserName** for the user and upon save, it is added to the in memory collection as shown in the following screenshot:



8. We define commands here to `SaveChanges` and `CancelChanges`, or even delete a particular item from the list.

```
private RegisterCommand _saveChanges;
public RegisterCommand SaveChanges
{

```

```

        get
        {
            this._saveChanges = this._saveChanges ?? new
RegisterCommand(this.OnSaveChanges, this.ValidateData);
            return this._saveChanges;
        }
    }
    public RegisterCommand CancelChanges
    {
        get { return new RegisterCommand(e => this.Parent.Close =
true); }
    }
    void OnSaveChanges(object state)
    {
        this.Parent.AddUser(this);
        this.FirstName = this.LastName = this.UserName = string.Empty;
    }
    bool ValidateData(object sender)
    {
        return !string.IsNullOrEmpty(this.UserName);
    }

    private RegisterCommand _delete;
    public RegisterCommand Delete
    {
        get
        {
            this._delete = this._delete ?? new RegisterCommand(e =>
this.Parent.DeleteUser(this));
            return this._delete;
        }
    }
}

```

The preceding code defines three commands. We are using the `RegisterCommand` class to define a command. I have already showed that there are two delegates that need to be passed to ensure the command is registered. The first one is the delegate that will get executed when the command is hit, and another is to validate the command. In runtime, when `ValidateData` returns false or `UserName` is blank, the button to which the command is bound is disabled. When the data is valid, the object is added to `ObservableCollection`.



It is important to note that `ObservableCollection` supports `INotifyCollectionChanged`, which is hooked by the controls to update the user interface when an object is added or deleted from the collection. If you use `List` or any other collection instead, the UI will not get refreshed automatically when you delete or add items to the collection.

9. The Command binding also allows you to refresh the button context by calling the `CanExecute` (or `ValidateData`) method again by raising the event `CanExecuteChanged`. Thus if we see the property `UserName`, it will look like:

```
string username;
public string UserName
{
    get { return this.username; }
    set
    {
        var username = this.username;
        this.username = value;
        this.RaisePropertyChanged<string>("UserName", username,
value);
        this.SaveChanges.OnCanExecuteChanged();
    }
}
```

Here in the preceding property, `PropertyChanged` is raised to ensure that the data in `UserName` reflects the `UIElement`. `CanExecuteChanged` is also invoked when the data is set to ensure that the validation block is re-executed (if any) such that the button that has command associated with it, re-enable it based on the returned value.

10. The `PropertyChanged` event needs to be raised when we need to update a property to the UI or vice versa. In the UI we define a calculated field with `FullName` which returns the combination of `FirstName` and `LastName`.

```
public string FullName
{
    get { return string.Format("{0} {1}", this.FirstName, this.
LastName); }
}
```

Now when the UI gets loaded, the property `FullName` gets empty value. We need to reload the value for the `FullName` when either the `FirstName` or the `LastName` gets changed. To ensure the `FullName` gets re-evaluated, we define the `FirstName` with a property raise call to `FullName`.

```
string firstname;
```

```
public string FirstName
{
    get { return this.firstname; }
    set
    {
        var name = this.firstname;
        this.firstname = value;
        this.RaisePropertyChanged<string>("FirstName", name,
value);
    }
}
```

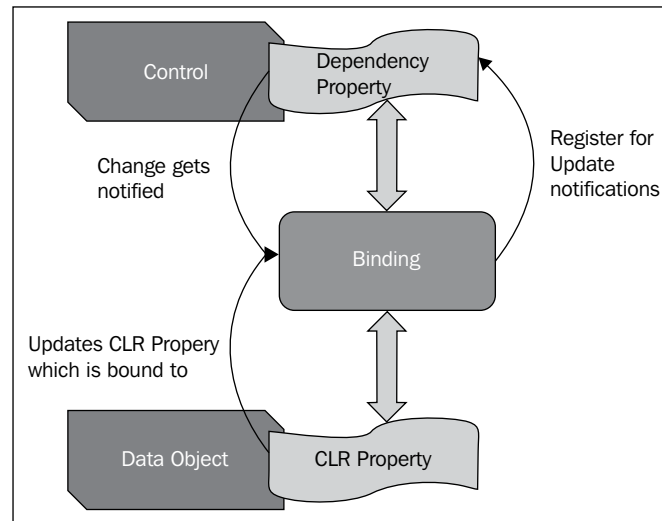
Here in the preceding code, `RaisePropertyChanged` raises not only the `PropertyChanged` for `FirstName` but also raises it for `FullName`, such that the value for `FullName` gets reevaluated and reflected to the UI control to which it is bound.

How it works...

MVVM architecture is built and supported by the WPF from its core by defining special functionalities to the controls. Each control in WPF is created deriving from `DependencyObject` which holds a number of `DependencyProperty` attributes. The WPF property system supports `DataBinding` inherently and does all the plumbing work that is required for `DataBinding` like:

- ▶ Hooking up with the `PropertyChanged` event for object if it inherits from `INotifyPropertyChanged`.
- ▶ Hooking up with the `CollectionChanged` event when working with a collection and data templates.
- ▶ Invoking change notification to the objects automatically from the UI.
- ▶ Determining the value of the property at runtime by determining the latest value of the object.
- ▶ Supporting property inheritance, final value resolution, and default value.
- ▶ Reducing memory footprints as the Property System maintains one value for all controls throughout if not changed. (For example: There will have one `Background` property set to red throughout the project if not any particular background of a `TextBlock` is changed.)
- ▶ Coercing property value based on runtime information.

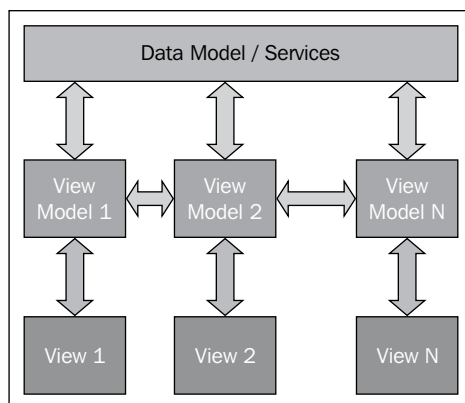
Hence, it is not just a property with backing up field and Binding can be associated with these properties in XAML to communicate between data objects. The `Dependency` property system also supports a property attached to a type which does not know anything about it.



In the preceding figure it has been demonstrated how the `DependencyProperty` system works. The Data Object is bound to the `Dependency` property using the **Binding** markup extension. As `DependencyProperty` itself is capable of providing change notifications, the `Binding` bounds to the property and registers to the event to get notifications from the control. When the object receives notification, it updates the CLR property for the object which is passed to `Binding`. Similarly, `Binding` also subscribes to change notifications of Data Object, such that when the update is notified from the CLR, it notifies the `Binding` to update the `Control` as well.

The `Dependency` property system is a unified model for any properties that takes part in any control. Thus the change notifications from the control can also update properties of other controls (for instance, the `FullName` property in our case is bound to some other property, but still it can notify the changes from another CLR property to update the user interface).

The MVVM model takes help from the property system to implement the interface between the View models and the View. The separation of concerns are maintained in such a way that the View Model or the Presentation logic can be re-used to different form factors too. For instance, you can implement a **ViewModel** for a Windows desktop and implement the UI for tabs, phones, and so on and re-use the same code. Visual Studio also comes with a new standard of portable class library, which can be used to deploy the same code targeting more than one CLR. This can be used to implement ViewModels and can be ported into any device.



The main intention of MVVM is to introduce a separation of View with the Presentation logic. Each View in the pattern defines its own ViewModel which bridges the DataModel and services layer. The ViewModel can refer to the data models and transfer data in a presentable form to the Views. Similarly, the ViewModels can also communicate with each other in the same way by which the Views of the application communicates, but here we do not communicate between Views rather we communicate using the ViewModels.

There's more...

MVVM needs lot of plumbing technology that is available with WPF, which enables the development of MVVM architecture very easily. Let us define these concepts a little further so that you have enough knowledge about these technologies.

What is DependencyProperty, and how to declare and use it?

WPF comes with a completely new technique for defining a property of a control. The unit of the new property system is a `Dependency` property, and the wrapper class which can create a `DependencyProperty` is called a `DependencyObject`. We use to register a `DependencyProperty` in the property system to ensure that the object contains the property in it and we can easily get or set the value of those properties whenever we like. We even use the normal CLR property to wrap around a `DependencyProperty` and use `GetValue` and `SetValue` to get and set values passed within it. To work with the `Dependency` property, you must derive the class from `DependencyObject`, as the whole observer which holds the new property system is defined within `DependencyObject`.

As a matter of fact, the `DependencyProperty` has lots of advantages over the normal CLR property. Let's discuss the advantages a bit before we create our own `DependencyProperty`:

- ▶ **Property Value Inheritance:** By Property Value Inheritance we mean that value of a `DependencyProperty` can be overridden in the hierarchy in such a way that the value with highest precedence will be set ultimately.
- ▶ **Data Validation:** We can impose Data Validation to be triggered automatically whenever the property value is modified.
- ▶ **Participation in Animation:** The `DependencyProperty` can be used for animation. WPF animation has capabilities to change value at an interval. By defining a `DependencyProperty`, you can eventually support animation for that property.
- ▶ **Participation in Styles:** Styles are elements that define the control. We can use Style Setters on the `DependencyProperty`.
- ▶ **Participation in templates:** Templates are elements that define the overall structure of the element. By defining the `DependencyProperty`, we can use it in templates.
- ▶ **DataBinding:** As each of the `DependencyProperty` itself invokes `INotifyPropertyChanged` whenever the value of the property is modified, `DataBinding` is supported internally. To read more about `INotifyPropertyChanged`, please read.
- ▶ **Callbacks:** You can have callbacks to a dependency property, so that whenever a property is changed, a callback is raised.
- ▶ **Resources:** A `DependencyProperty` can take a resource. So in XAML, you can define a resource for the definition of a `DependencyProperty`.
- ▶ **Metadata overrides:** You can define certain behaviors of a `DependencyProperty` using `PropertyMetadata`. Thus overriding a metadata from a derived property will not require you to redefine or re-implementing the whole property definition.
- ▶ **Designer Support:** A `DependencyProperty` gets support from Visual Studio Designer. You can see all the dependency properties of a control listed in the **Property Window** of the Designer.

In these, some of the features are only supported by the `DependencyProperty`. Animation, Styles, Templates, Property Value Inheritance, and so on could only be participated using the `DependencyProperty`. If you use CLR property instead in such cases, the compiler will generate error.

To define a dependency property you need to register it to the `DependencyProperty` system first.

```
public static readonly DependencyProperty MyCustomProperty
= DependencyProperty.Register("MyCustom", typeof(string),
    typeof(Window1));

public string MyCustom
```

```

{
    get
    {
        return this.GetValue(MyCustomProperty) as string;
    }
    set
    {
        this.SetValue(MyCustomProperty, value);
    }
}

```

In the preceding code, I have simply defined a `DependencyProperty`. You must have been surprised that a `DependencyProperty` was declared as static. Yes, like you even I was surprised when I first saw that. But later on after reading about the `Dependency` property, I came to know that a `DependencyProperty` is maintained at class level, so you may want `Class A` to have a property `B`. So property `B` will be maintained to all the objects that `Class A` has individually. The `DependencyProperty` thus creates an observer for all those properties maintained by `Class A` and stores it there. Thus, it is important to note that a `DependencyProperty` should be maintained as static.

The naming convention of a dependency property states that it should have the same wrapper property which is passed as the first argument. Thus in our case, the name of the Wrapper `MyCustom`, which we will use in our program, should be passed as the first argument of the register method. Also, the name of the `DependencyProperty` should always be suffixed with the property of the original Wrapper key. So in our case the name of the `DependencyProperty` is `MyCustomProperty`. If you don't follow this, some of the functionality will behave abnormally in your program.

It should also be noted that you should not write your logic inside the Wrapper property, as it will not be called every time the property is called for. It will internally call `GetValue` and `SetValue` itself. So if you want to write your own logic when the dependency property is fetched, there are callbacks to do them.

The `MyCustom` dependency property now enables `DataBinding`, `Animation`, `Styles` and all other dependency property system enhancements.

After defining the simplest `DependencyProperty` ever, let's make it a little enhanced. To add metadata for a `DependencyProperty`, we use the object of the class `PropertyMetadata`. If you are inside a `FrameworkElement` as I am inside a `UserControl` or a `Window`, you can use `FrameworkMetadata` rather than `PropertyMetadata`. Let's see how to code :

```

static FrameworkPropertyMetadata propertymetadata = new FrameworkPro
pertyMetadata("Comes as Default", FrameworkPropertyMetadataOptions.
BindsTwoWayByDefault | FrameworkPropertyMetadataOptions.
Journal, new PropertyChangedCallback(MyCustom_PropertyChanged), new
CoerceValueCallback(MyCustom_CoerceValue),
false, UpdateSourceTrigger.PropertyChanged);

```

```
public static readonly DependencyProperty MyCustomProperty
= DependencyProperty.Register("MyCustom", typeof(string),
typeof(Window1),
propertymetadata,      new ValidateValueCallback(MyCustom_Validate));

private static void MyCustom_PropertyChanged(DependencyObject dobj,
DependencyPropertyChangedEventArgs e)
{
    //To be called whenever the DP is changed.
    MessageBox.Show(string.Format("Property changed is fired : OldValue
{0} NewValue : {1}", e.OldValue, e.NewValue));
}

private static object MyCustom_CoerceValue(DependencyObject dobj,
object Value)
{
    //called whenever dependency property value is reevaluated. The
return value is the
    //latest value set to the dependency property
    MessageBox.Show(string.Format("CoerceValue is fired : Value {0}",
Value));
    return Value;
}

private static bool MyCustom_Validate(object Value)
{
    //Custom validation block which takes in the value of DP
    //Returns true / false based on success / failure of the
validation
    MessageBox.Show(string.Format("DataValidation is Fired : Value
{0}", Value));
    return true;
}

public string MyCustom
{
    get
    {
        return this.GetValue(MyCustomProperty) as string;
    }
    set
    {
        this.SetValue(MyCustomProperty, value);
    }
}
```

So this is little more elaborate. We define `FrameworkMetaData`, where we have specified `DefaultValue` for the `DependencyProperty` as "Comes as Default", so if we don't reset the value of the `DependencyProperty`, the object will have this value as default. `FrameworkPropertyMetaDataOption` gives you a chance to evaluate the various metadata options for the dependency property. Let's see the various options for the enumeration.

- ▶ `AffectsMeasure`: This invokes `AffectsMeasure` for the layout element where the object is placed.
- ▶ `AffectsArrange`: This invokes `AffectsArrange` for the layout element.
- ▶ `AffectsParentMeasure`: This invokes `AffectsMeasure` for the parent.
- ▶ `AffectsParentArrange`: This invokes `AffectsArrange` for the parent control.
- ▶ `AffectsRender`: This renders the control when the value is modified.
- ▶ `NotDataBindable`: Data binding could be disabled.
- ▶ `BindsWithTwoWayByDefault`: By default data binding will be one way. If you want your property to have a two way default binding, you can use this.
- ▶ `Inherits`: This ensures that the child control inherits values from its base.

You can use more than one option using `|` separation as we do for flags.

`PropertyChangedCallback` is called when the property value is changed. So it will be called after the actual value is modified already. `CoerceValue` is called before the actual value is modified. That means after `CoerceValue` is called, the value that we return from it will be assigned to the property. The validation block will be called before `CoerceValue`, so this event ensures whether the value passed in to the property is valid or not. Depending on the validity, you need to return true or false. If the value is false, the runtime generates an error.

So in the preceding application after you run the code, `MessageBox` comes up for the following:

- ▶ `ValidateCallback`: You need to put logic to validate the incoming data as the `Value` argument. True makes it take the value, false will throw the error.
- ▶ `CoerceValue`: You can modify or change the value depending on the value passed as argument. It also receives `DependencyObject` as an argument. You can invoke `CoerceValueCallback` using the `CoerceValue` method associated with `DependencyProperty`.
- ▶ `PropertyChanged`: This is the final `Messagebox` that you see, which will be called after the value is fully modified. You can get the `OldValue` and `NewValue` from the `DependencyPropertyChangedEventArgs`.

Another additional benefit of the dependency property is that it can attach itself to a control other than where it is defined to. Attached property enables you to attach a property to an object that is outside the object altogether, making it define a value for it using that object.

Let's declare an attached `DependencyProperty`:

```
public static readonly DependencyProperty IsValuePassedProperty =
    DependencyProperty.RegisterAttached("IsValuePassed", typeof(bool),
        typeof(Window1),
            new FrameworkPropertyMetadata(new PropertyChangedCallback(
                IsValuePassed_Changed)));
public static void SetIsValuePassed(DependencyObject obj, bool value)
{
    obj.SetValue(IsValuePassedProperty, value);
}

public static bool GetIsValuePassed(DependencyObject obj)
{
    return (bool)obj.GetValue(IsValuePassedProperty);
}
```

Here I have declared a `DependencyObject` that holds a value `IsValuePassed`. The object is bound to `Window1`, so you can pass a value to `Window1` from any `UIElement`. `SetIsValuePassed` and `GetIsValuePassed` are the method stubs that are used to get the data from the dependency property matrix. `FrameworkPropertyMetadata` here allows you to specify a `PropertyChanged` callback.

So in my code, `UserControl` can pass the value of the property to the window.

```
<local:MyCustomUC x:Name="ucust" Grid.Row="0" local:Window1.
    IsValuePassed="true"/>
```

You can see in the preceding code that `IsValuePassed` can be set from an external user control, and the same will be passed to the actual window.

As you can see I have added two static methods to individually set or get values from the object. This would be used from the code to ensure that we pass the value from code from appropriate objects.

Say you add a button and want to pass values from code. In such cases the static method will help.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Window1.SetIsValuePassed(this, !(bool)this.GetValue(IsValuePassedProperty));
}
```

A dependency property is a bundle of interesting features in WPF without which, the whole system of data binding wouldn't run.

What are the attributes of Binding and use of Binding Expression?

Binding is the most important feature of WPF that makes it different from the rest of the technology. It is a framework of types that runs over the dependency property system, and every control that defines a set of dependency properties can actually hook Binding to it and create special bonding between two or more objects. `DataBinding` is not new to .NET technology. `DataBinding` was present before the introduction of WPF. In ASP.NET, we bind data elements to render proper data from the control. We generally pass in `DataTable` and bind the templates to get data from individual `DataRow`s. On the other hand, in the case of traditional windows forms application, we can also bind a property with a data element. Bindings can be added to properties of objects to ensure that whenever the property changes value, the data is internally reflected to the data. So in one word, `DataBinding` is nothing new to the system. The main objective of `DataBinding` is to show data to the application and hence, reduce the amount of work the application developer needs to write to just make the application properly display the data. WPF `DataBinding` is not just simple `DataBinding` that is supported by other technology and the `Binding` type is not inbuilt into the controls, rather, WPF `DataBinding` is a separate concern that runs homogenously into the system and can take part with one control to the other or an object outside the WPF world. When you bind one property with another, the framework automatically does the plumbing to ensure that when one object changes its state, the other gets notified.

```
<TextBlock Text="{Binding Name}" />
```

The preceding line of code binds the `Name` property with the `Text` property of the `TextBlock` object where `Name` is defined on a type that has been set to `DataContext` of the object.

Binding is actually a Markup Extension. It is a type newly introduced called `Binding`. Like other classes, `Binding` also exposes few properties. Let's discuss them:

- ▶ **Source:** The source property holds the data source. By default, it references `DataContext` of the control. If you place the `Source` property for the `Binding`, it will take that in lieu of the original `DataContext` element.
- ▶ **ElementName:** In case of `Binding` with another element, `ElementName` takes the name of the elements defined within the XAML for reference of the object. `ElementName` acts as a replacement to `Source`. If a path is not specified for the `Binding`, it will use `ToString` to get the data from the object passed as `Source`.
- ▶ **Path:** Path defines the actual property path to get the string data. If the end product is not a string, it will also invoke `ToString` to get the data.
- ▶ **Mode:** It defines how the data will be flown. `OneWay` means object will be updated only when source is updated, on the contrary `OneWayToSource` is the reverse. `TwoWay` defines the data to be flown in both ways.

- ▶ **UpdateSourceTrigger:** This is another important part of any Binding. It defines when the source will be updated. The value of `UpdateSourceTrigger` can be:
 - ❑ **PropertyChanged:** It is the default value. As a result, whenever anything is updated in the control, the other bound element will reflect the same.
 - ❑ **LostFocus:** It means whenever the property loses its focus, the property gets updated.
 - ❑ **Explicit:** If you choose this option, you need to explicitly set when to update the Source. You need to use `UpdateSource` of `BindingExpression` to update the control.

```
BindingExpression bexp = mytextbox.GetBindingExpression("txtName",  
    TextBox.TextProperty);  
bexp.UpdateSource();
```

By this, the source gets updated for a type `txtName` within the solution.

- ▶ **Converter:** Converter gives you an interface to put an object which will be invoked whenever the Binding objects get updated. Any object that implements `IValueConverter` can be used in place of Converter. Refer to <http://bit.ly/WPFBindingConverter> for further details.
- ▶ **ConverterParameter:** It is used in addition to Converter to send parameters to Converter.
- ▶ **FallbackValue:** Defines the value which will be placed whenever the Binding cannot return any value. By default, it is blank.
- ▶ **StringFormat:** A formatting string that indicates the Format to which the data will follow.
- ▶ **ValidatesOnDataErrors:** When specified, the data errors will be validated. You can use `IDataErrorInfo` to run your custom validation block when Data object is updated. We will discuss more about it in the next part.
- ▶ **ValidatesOnNotifyDataError:** Same as the other one, but in this case the `INotifyDataErrorInfo` is taken for granted and works asynchronously.
- ▶ **Delay:** WPF also supports a property `Delay` to asynchronously make the binding notification delayed for a while. As Binding can generate a lot of data update in the chain, it is important to have a `Delay` property for Binding so that the data chain does not get updated when values get changed frequently.

Now as I told you, Binding is a new type that runs outside the existing application and hooks itself to any environment and binds two properties which are different in all respect. For instance, we can bind object A with object B such that:

- ▶ A is a WPF Control, B is another WPF control
- ▶ A is a WPF Control, B is a CLR class
- ▶ A is a WPF Control and B is a Static object

Binding also supports static objects. To define a static object we need to create a static event and raise the event when the object changes its state.

```
private static Color _color;
public static Color Color
{
    get { return _color; }
    set
    {
        _color = value;
        RaiseColorChanged();
    }
}

public static event EventHandler ColorChanged;
public static void RaiseColorChanged()
{
    EventHandler ohandler = ColorChanged;
    if (ohandler != null)
        ohandler(null, EventArgs.Empty);
}
```

In the preceding code, `Color` is a static property and we track the change notification of the `Color` property using separate `ColorChangedEvent`. To bind the `Color` property with the object we use:

```
<TextBlock>
  <TextBlock.Background>
    <SolidColorBrush Color="{Binding (local:ColorHelper.Color)}"/>
  </TextBlock.Background>
</TextBlock>
```

Here in `ColorHelper.Color` is bound to the `Background` property of `TextBlock`. `Color` can be changed in the background to have this reflected to `TextBlock` as well.

There is also a notion of `BindingGroup` which allows a way to create relationship between Bindings. Generally, `BindingGroup` is used for validation such that when two bound objects updates their value it can validate and have errors notified.

```
<StackPanel.BindingGroup>
  <BindingGroup NotifyOnValidationError="True">
    <BindingGroup.ValidationRules>
      <local:ValidateDateAndPrice ValidationStep="CheckData" />
    </BindingGroup.ValidationRules>
  </BindingGroup>
</StackPanel.BindingGroup>
```


In the preceding code, `BindingGroup` is mentioned for `StackPanel`, which validates using `ValidationRules` setup for the application. `ValidateDateAndPrice` is a type defined within the application with `ValidationStep` defined. The type is derived from `ValidationRule` to ensure that the object calls the `Validate` method overridden on it.

It is important to remember that sometimes it becomes essential to get information about Binding from the code itself. We can use the `BindingExpression` type which evaluates Binding to get information about it.

```
BindingExpression bexp = mytextbox.GetBindingExpression("txtName",
    TextBox.TextProperty);
```

The object `bexp` can get information about the Binding like:

```
//The target
DependencyObject target = bexp.Target;

//The target property
DependencyProperty targetProperty = bexp.TargetProperty;

//The source object
object source = bexp.ResolvedSource;

//The source property name
string sourcePropertyName = bexp.ResolvedSourcePropertyName;

//The binding group
BindingGroup bindingGroup = bexp.BindingGroup;

//The binding group's owner
if (bindingGroup != null)
{
    DependencyObject bindingGroupOwner = bexp.BindingGroup.Owner;
}
```

Thus, you can get information about `Target`, `Source`, and other information anywhere when you need.

How to implement data validation blocks in MVVM?

As we move ahead with WPF, we find lots of flexibilities that WPF provides for UI development. One of such is to define a validation rule to work with controls bound with data elements, such that the data gets annotated automatically using styles when validation rule fails. In this article, I will discuss how easily you can impose custom validation to your data elements by doing the entire thing from within the class itself.

First let us create `DataElement` for which I need to use data validation. As we know Binding needs few things like `INotifyPropertyChanged` event to handle binding, it needs `IDataErrorInfo` to handle `ValidatesOnDataErrors` calls. So the first thing that you need to do is to create a class from the `IDataErrorInfo` interface.

```
public class Contact : IDataErrorInfo
{
    private string _Name;
    private string _desig;

    public string Name
    {
        get { return _Name; } set { _Name = value; }
    }

    public string Designation
    {
        get { return _desig; } set { _desig = value; }
    }

    #region IDataErrorInfo Members

    public string Error
    {
        get { return "Error occurred"; }
    }

    public string this[string columnName]
    {
        get
        {
            return this.GetResult(columnName);
        }
    }

    private string GetResult(string columnName)
    {
        PropertyInfo info = this.GetType().
        GetProperty(columnName);
        if (info != null)
        {
            string value= info.GetValue(this, null) as string;
            if (string.IsNullOrEmpty(value))
                return string.Format("{0} has to be set", info.
Name);
            else if(value.Length < 5)

```

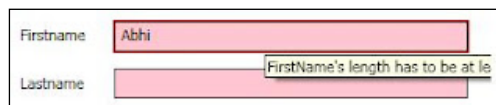
```
        return string.Format("{0}'s length has to be at  
least 5 characters !", info.Name);  
    }  
    return null;  
}  
#endregion  
}
```

Now this interface automatically calls the `GetResult` method on every call to property changes and thus it gets validated. The error is a `Validation` property that represents a collection of error contents. Now to ensure that we visually show the errors to the user, we need to apply styles which triggers on the `Validation.HasError` property.

```
<Style TargetType="{x:Type TextBox}">  
    <Style.Triggers>  
        <Trigger Property="Validation.HasError" Value="true">  
            <Setter Property="ToolTip" Value="{Binding  
RelativeSource={RelativeSource Self}, Path=(Validation.Errors)[0].  
ErrorContent}"/>  
            <Setter Property="Control.Background" Value="Pink"/>  
        </Trigger>  
    </Style.Triggers>  
</Style>
```

So if `TextBox` is bound to a property with `ValidatesOnDataErrors` set to `True`, it calls the `IDataErrorInfo` interface automatically and applies the style for the `DataBound` object.

```
<Window.Resources>  
    <AppCode:Contact x:Key="dsContact"/>  
</Window.Resources>  
<TextBox BorderBrush="Black" Margin="80,54.04,19,74.96" Height="22"  
    Name="txtLastName" Text="{Binding  
Source={StaticResource dsContact}, UpdateSourceTrigger='LostFocus',  
Path=LastName, ValidatesOnDataErrors=True}">
```



In the preceding screenshot, you can see that the `Firstname` and `Lastname` fields both shows a pink background which indicates that it has error.

Similar to `IDataErrorInfo`, WPF exposes another interface called `INotifyDataErrorInfo`, which additionally includes an event called `ErrorsChanged` and runs the validation logic asynchronously. The `ErrorsChanged` event needs to be raised when new errors or lack of errors are detected. We need to raise the event just like `PropertyChanged` on every property which might throw errors.

```
public string Name
{
    get { return _Name; }
    set { _Name = value; this.OnErrorsChanged("Name"); }
}
```

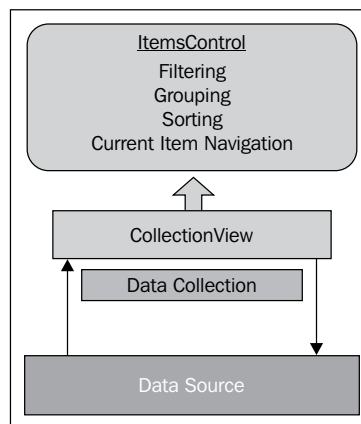
Here in the preceding code when `Name` is set, `OnErrorsChanged` is invoked that will call the `GetErrors` method, in which you can write the validation logic for the property.

How to work with `CollectionView` using Live Shaping?

While working with collection in WPF, a special type has been introduced which helps in data manipulation in the UI very easily. `CollectionView` in WPF allows the data to be sorted, grouped, filtered, and so on. It is a layer that runs over the data objects which allows you to define rules for sorting, filtering, grouping, and so on, and manipulate the display of data rather than modifying the actual data objects. Therefore in other words, `CollectionView` is a class which takes care of the View totally, and gives us the capability to handle certain features incorporated within it.

To specify `CollectionView` for a list, you can use:

```
this.ListBoxControl.ItemsSource = CollectionViewSource.
    GetDefaultView(this.Source);
```



`collectionView` is an abstraction to the data object, such that it supports features that developers often need.

Sorting

Sorting can be applied to `CollectionView` in a very easy way. You need to add `SortDescription` to `CollectionView`. `CollectionView` actually maintains a stack of the `SortDescription` objects, each of them being a structure can hold information of a column and the direction of Sorting. You can add them in `collectionView` to get the desired output.

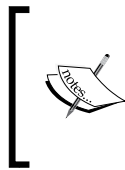
Say I store `CollectionView` as a property:

```
ICollectionView Source { get; set; }
```

Now you want to sort the existing collection:

```
this.Source.SortDescriptions.Add(new SortDescription("Name",  
ListSortDirection.Descending));
```

Hence `CollectionView` will be sorted based on Name and in descending order.



The default behavior of `CollectionView` automatically gets refreshed when new `SortDescription` is added to it. For performance issue, you might use `DeferRefresh()` if you want to refresh only once to add `SortDescription` more than once.

Grouping

You can create a custom group for `ICollectionView` in the same way as you do for Sorting. To create a group of elements, you need to use `GroupStyle` to define the template for the group and to show the name of the group in group header.

```
<ListView.GroupStyle>  
  <GroupStyle>  
    <GroupStyle.HeaderTemplate>  
      <DataTemplate>  
        <TextBlock Text="{Binding Name}" />  
      </DataTemplate>  
    </GroupStyle.HeaderTemplate>  
  </GroupStyle>  
</ListView.GroupStyle>
```

Here we define the group `HeaderTemplate` for each group, so that `TextBlock` shows the name of the group item by which the grouping is made. You can specify more than one grouping information for a single collection. To group a collection, you need to use:

```
this.Source.GroupDescriptions.Add(new PropertyGroupDescription("Depart  
ment"));
```



Grouping turns off virtualization. So if you are dealing with large amounts of data, grouping may lead to performance issue.

You can apply custom grouping as well by defining `IValueConverter` in `PropertyGroupDescription` as well.

Filtering

Filtering requires a delegate (`Predicate`) based on which the filter will occur. `Predicate` takes in the item and based on the value (true or false) it returns, it selects or unselects an element.

```
this.Source.Filter = item =>
{
    ViewItem vitem = item as ViewItem;
    if (vitem == null) return false;

    return vitem.Name.Contains("A");
};
```

This will select only the elements which have A in their names.

Current Record Manipulation

`ICollectionView` also allows to synchronize items with the current position of the element in `CollectionView`. Each `ItemsControl`, which is the base class of any `ListControl` in WPF, exposes a property called `IsSynchronizedWithCurrentItem`, when set to true will automatically keep the current position of the `CollectionView` in sync.

There are methods like:

```
this.Source.MoveCurrentToFirst();
this.Source.MoveCurrentToPrevious();
this.Source.MoveCurrentToNext();
this.Source.MoveCurrentToLast();
```

These allow you to navigate around `CurrentItem` of `CollectionView`. You can also use the `CurrentChanged` event to intercept your selection logic around the object.

Live Shaping

When we need live data to be sorted while some data is being inserted on the fly, the Live Shaping technique comes in very handy. With the Live Shaping technique you can activate live sorting, live grouping, and live filtering separately based on their use.

For LiveShaping, we use `ICollectionViewLiveShaping` instead of `ICollectionView`. `GetCollectionView` returns `ICollectionViewLiveShaping` rather than `ICollectionView` in the latest release of .NET. Hence, you can just cast the object of `ICollectionView` to `ICollectionViewLiveShaping` to configure it when required.

We can use the same `CollectionView` class to apply Filtering, Sorting, or Grouping. To do this, you can use:

```
var shaping = this.Source as ICollectionViewLiveShaping;
shaping.IsLiveFiltering = true;
shaping.LiveFilteringProperties.Add("Name");
shaping.IsLiveSorting = true;
shaping.LiveSortingProperties.Add("Name");
shaping.IsLiveGrouping = true;
shaping.LiveGroupingProperties.Add("Department");
```

Here in the preceding code, we specify the LiveShaping properties for `CollectionView` which is by default turned off. In this case the `LiveFiltering`, `LiveSorting`, and `LiveGrouping` are turned on. Turning these on in the `CollectionView` for a large data collection can produce performance implications, so it is recommended to use them only when it is required.

See also

- ▶ Refer to the following link:

<http://bit.ly/WPFMVVM>

Using the Ribbon User Interface in WPF

Ribbon-based user interface is the latest trend in designing Windows-based applications. The latest UI trends are inclined towards Ribbon-based windows. The modern operating system is promoting Ribbon as the next generation UI for Windows. You can read more about Ribbon from the link [http://en.wikipedia.org/wiki/Ribbon_\(computing\)](http://en.wikipedia.org/wiki/Ribbon_(computing)).

WPF applications support the usage of ribbons. The base class library introduced a number of APIs that draws Ribbon Window UI on the screen.

In this recipe, we are going to introduce the Ribbon, and also show how these components can be used in normal scenarios as well as using MVVM patterns.

Getting ready

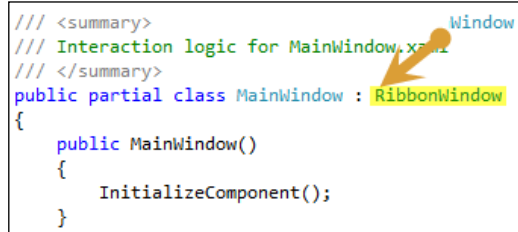
Before starting the recipe, let us open Visual Studio IDE and create a WPF application. Currently as I am using Visual Studio RC, there is Visual Studio template that directly creates a `RibbonWindow` application with the appropriate `.dll` files that we need for the application. So we do it manually.

After the initial application has been loaded to the IDE, we add a reference of "`System.Windows.Controls.Ribbon.dll`" to the application. This `dll` contains all the important APIs that are needed to create a Ribbon-based application. You should notice that the application is loaded with one window created, configured, and named as `MainWindow.xaml`. If you open the XAML file, it shows the `Window` tag that forms the window for the WPF application.

Ribbons are actually windows with the tag `RibbonWindow`. Change the `Window` tag to `RibbonWindow` to convert the appearance of `MainWindow` to `RibbonWindow`. We also change the background code from `Window` to `RibbonWindow`. The code looks like:

```
<RibbonWindow x:Class="RibbonSample.MainWindow"
              xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
              Title="MainWindow" Height="350" Width="525">
  <Ribbon>
  </Ribbon>
</RibbonWindow>
```

The code after changing the inherited class from `Window` to `RibbonWindow` it looks like:



```
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : RibbonWindow
{
    public MainWindow()
    {
        InitializeComponent();
    }
}
```

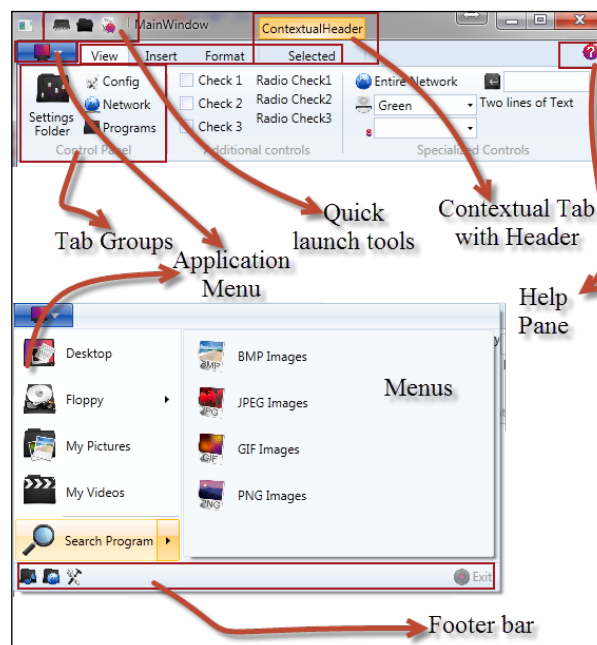

You can see `MainWindow` after changing the code inherits from `RibbonWindow` rather than `Window`. Also, do not forget to add the namespace using `System.Windows.Controls.Ribbon;`. Now run the application.



The window shows a blank ribbon area where we can add controls. The whole ribbon window is categorized into a number of sections, each of which forms a special section of the window and is used for special purposes:

- ▶ Ribbon window
- ▶ Quick launch actions
- ▶ Ribbon menu
- ▶ Ribbon tabs
- ▶ Contextual tabs

If we look into the actual UI for each of the sections of the window, we figure out like the one shown in the following screenshot:



In the preceding screenshot, we depicted all the sections of the ribbon window with **Application Menu**. These commands in fact inherited from the regular controls, and support most of the actions that are supported by the normal controls. Each of these controls supports command pattern as well as events. Hence, if you are following the MVVM pattern to create your application, you have total support if it is for every control as well. Now let us start coding the application.

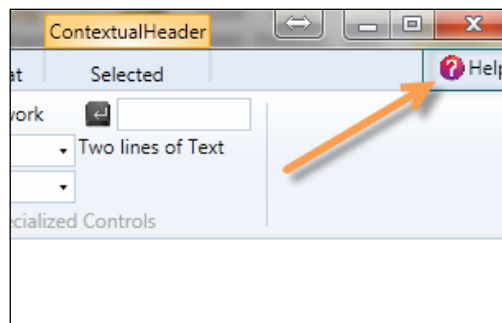
How to do it...

Now, let us create a ribbon application to test the elegance and styles that you can incorporate inside your applications.

1. Add a ribbon to the ribbon window. Each window can associate with only one ribbon.
2. A ribbon contains a **Title** (which defines the title of the ribbon), **Application Menu**, **Quick Access Toolbar**, **Contextual Tab Group**, and **Help Pane**. The sections are depicted in the previous figure. Let us now add each of these sections and define each of them one by one.
3. At the very right corner of the screen just below the Close button of the window, we have Help Pane. The Help Pane is a Content Control that has been defined to manipulate this section. Let us add a `HelpButton` inside the `HelpPaneContent` which is shown in the following screenshot:

```
<Ribbon.HelpPaneContent>
    <RibbonButton SmallImageSource="Images\Helpandsupport.ico"
    Label="Help"/>
</Ribbon.HelpPaneContent>
```

The preceding code will add a small button on the right-hand side on top of the ribbon:



This section supports only one control to be added. It shows this control on the top right-hand corner of the ribbon. As you can see, the button appears with the label and the `SmallImage`. Each ribbon-based control contains two `ImageSource` properties (that is, `SmallImageSource` and `LargeImageSource`) which is used to define the image for the control and automatically changes the image based on the size of the control.

1. Each `RibbonWindow` has an associated `ApplicationMenu`. On the top-left corner of the ribbon, you will see a big button which opens up a pop-up menu for the application. It is important to choose the most important menus that needs to be used by the users. Generally, the user looks for an application menu when he needs to work on some selected special commands. `ApplicationMenu` is composed of `RibbonApplicationMenuItem` and `RibbonApplicationSplitMenuItem`. The former creates a single menu item for the application menu which either has submenus or represents the leaf menu item, but the latter represents a `RibbonApplicationSplitButton` which has a separator between the actual menu and the submenu from it:

```
<Ribbon.ApplicationMenu>
    <RibbonApplicationMenu SmallImageSource="Images\MyComputer.
ico">

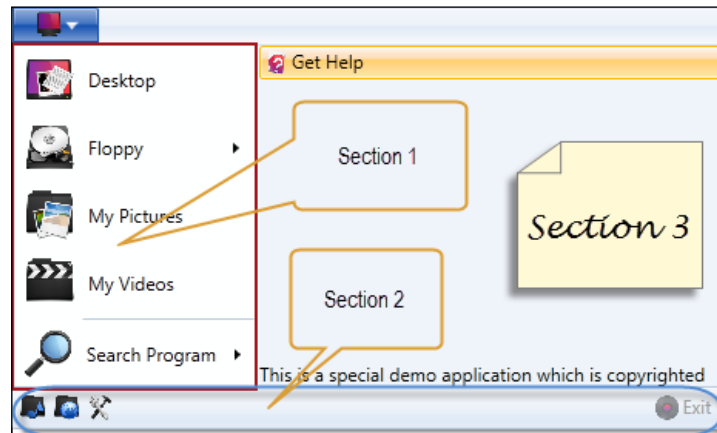
        <RibbonApplicationMenuItem Header="Desktop"
ImageSource="Images\Desktop.ico" KeyTip="D" Click="RibbonApplicati
onMenuItem_Click"/>

        <RibbonSeparator />
        <RibbonApplicationSplitMenuItem Header="Search Program"
ImageSource="Images\Search.ico" Click="RibbonApplicationMenuIt
em_Click">

            <RibbonApplicationMenuItem Header="BMP Images"
ImageSource="Images\BMPImage.ico" Click="RibbonApplicationMenuIt
em_Click"/>
        </RibbonApplicationSplitMenuItem>
        <RibbonApplicationMenu.FooterPaneContent>

            </RibbonApplicationMenu.FooterPaneContent>
            <RibbonApplicationMenu.AuxiliaryPaneContent>
                <TextBlock Text="This is the Auxilary Pane. You can
also add RibbonGallery to AuxiliaryPane." />
            </RibbonApplicationMenu.AuxiliaryPaneContent>
        </RibbonApplicationMenu>
    </Ribbon.ApplicationMenu>
```

The preceding code creates an application menu for the ribbon we have added to the window. `ApplicationMenu` is the only menu that can be associated with `RibbonWindow`.



The preceding screenshot shows the Application Menu. Each ribbon window can contain a single application menu which can be opened by clicking on the top-left section of the ribbon. The **RibbonApplicationMenu** is divided into three parts:

- ❑ **RibbonApplicationMenu:** This section shows all the menu items that we add to the Application Menu. In the preceding screenshot, the Application Menu forms the left-hand side of the screen and is marked as **Section 1**.
 - ❑ **FooterPane:** Each Application Menu can have a **Footer Pane**. This is a normal **ContentControl** which is capable of showing any WPF content. We have placed a few buttons as **QuickLinks** for the **Application Menu**. The section is marked as **Section 2**.
 - ❑ The **AuxiliaryPaneContent** is on the right-hand side of the Application Menu where we generally show auxiliary items related to the application. This section is marked as **Section 3** and is also a **ContentControl** and is flexible to show anything as content.
2. For every ribbon window, the application title bar has been utilized more efficiently to display a small set of controls which forms the Quick Access Toolbar for the window. We generally show the most common commands inside the Quick Access Toolbar. A Quick Access Toolbar is a collection of the `RibbonButton` items which shows only `SmallImage` to identify without considering the label for the button when added to the toolbar. The toolbar can show seven items at a time, and if more than seven buttons are added to the toolbar, it creates a menu automatically to show more controls:

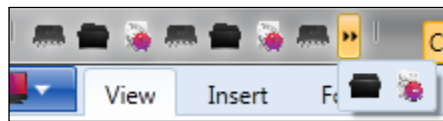
```
<Ribbon.QuickAccessToolBar>
    <RibbonQuickAccessToolBar HorizontalAlignment="Right"
    IsOverflowOpen="True">
```

```

        <RibbonButton SmallImageSource="Images\RAMDrive.ico"
KeyTip="R" Label="RAM"/>
        <RibbonButton SmallImageSource="Images\OpenFolder.ico"
KeyTip="O"/>
        <RibbonButton SmallImageSource="Images\MyRecentDocuments.
ico" KeyTip="R"/>
    </RibbonQuickAccessToolBar>
</Ribbon.QuickAccessToolBar>

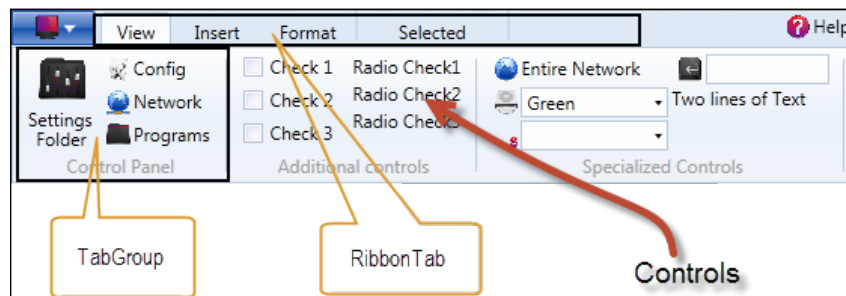
```

In the preceding code we have added a `RibbonQuickAccessToolBar` item to the `RibbonWindow`. The `RibbonButton` has specified its `SmallImageSource` and the `KeyTip`. The label will not show for the toolbar and it is immaterial to specify it.



The preceding screenshot shows the toolbar with the overflow button opened to show the menu of extra buttons that cannot fit into the toolbar.

- The most important section of the ribbon is a **RibbonTab**. Each Ribbon holds a collection of **RibbonTab**, while each Tab shows a menu to the user which is shown one at a time. A **RibbonTab** has a name specified as Header and a number of **RibbonGroup**, each Group can hold a collection of controls.



In the preceding recipe, the **RibbonTab** is an individual menu for the ribbon which is shown one at a time. In the preceding screenshot, the **RibbonTab** options are listed as **View**, **Insert**, **Format**, and so on. Each tab can hold a number of **TabGroup**. The **TabGroup** for the **View RibbonTab** are **Control Panel**, **Addition Controls**, and **Specialized Controls**:

```

<RibbonTab Header="View">
    <RibbonGroup Header="Control Panel">
        <RibbonButton LargeImageSource="Images\ControlPanel.ico"
            Label="Settings Folder" />
    </RibbonGroup>
</RibbonTab>

```

```

        <RibbonButton SmallImageSource="Images\
ConfigurationSettings.ico"
            Label="Config" />
        <RibbonButton SmallImageSource="Images\NetworkConnections.
ico"
            Label="Network"/>
        <RibbonButton SmallImageSource="Images\ProgramGroup.ico"
            Label="Programs"
        />
    </RibbonGroup>
</RibbonTab>

```

In the preceding code, we have added a `RibbonTab` to the `RibbonWindow` and specified few `RibbonGroup`. The `RibbonGroup` header is shown at the bottom of each group, and each group individually collapses or expands depending on the available space, automatically. A button has three states:

- ❑ Large (big image with text at the bottom of the image)
 - ❑ Medium (small image with text on right-hand side of the button)
 - ❑ Small (small image without text)
4. Even though we can add controls inside the `RibbonGroup`, it is recommended that you use specially designed ribbon controls only inside the ribbon. For instance, you should prefer `RibbonButton` over `Button`, or `RibbonCheckBox` over a normal `CheckBox`. The Ribbon based controls automatically style uniformly when placed inside the Ribbon. Some of the important `RibbonControls` are:

- ❑ `RibbonButton`
- ❑ `RibbonToggleButton`
- ❑ `RibbonComboBox`
- ❑ `RibbonTextBox`
- ❑ `RibbonCheckBox`
- ❑ `RibbonRadioButton`
- ❑ `RibbonTwoLineText`

The `RibbonTwoLineText` is a special control that shows text using **LineStacking strategy**.

5. A ribbon control also supports a `RibbonGallery`. A gallery is a specially designed collection of items that acts as a group of elements.

```

<RibbonComboBox Label="1"
    SmallImageSource="Images/DVDDrive.ico"
    IsEditable="True" >
    <RibbonGallery SelectedValue="Green"

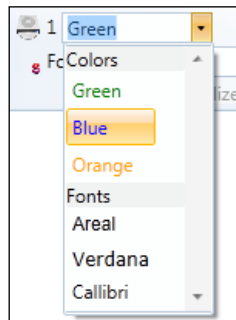
```

```

        SelectedValuePath="Content "
        MaxColumnCount="1">
        <RibbonGalleryCategory Header="Colors">
            <RibbonGalleryItem Content="Green" Foreground="Green"
/>
            <RibbonGalleryItem Content="Blue" Foreground="Blue" />
            <RibbonGalleryItem Content="Orange"
Foreground="Orange" />
        </RibbonGalleryCategory>
        <RibbonGalleryCategory Header="Fonts">
            <RibbonGalleryItem Content="Areal" FontFamily="Arial"
/>
            <RibbonGalleryItem Content="Verdana"
FontFamily="Verdana" />
            <RibbonGalleryItem Content="Callibri"
FontFamily="Callibri" />
        </RibbonGalleryCategory>
    </RibbonGallery>
</RibbonComboBox>

```

In the preceding design, we have specified a gallery of items inside RibbonCombobox. A RibbonGalleryItem is individual items for a Ribbon which can be selected as a ComboBox item. RibbonGalleryItem supports grouping. RibbonGalleryCategory specifies the category for each gallery group. Here in the preceding code we have created two groups of gallery one defining the **Colors** and another defining the **Fonts**.



MaxColumnCount specifies the number of columns that the gallery will show on the ribbon.

6. Each Ribbon can specify `ContextualTabGroups`. A `ContextualTabGroup` specifies a single tab inside the ribbon which is shown based on some context. For instance, say you want to show a special tab when some items inside the window are selected.

```
<RibbonTab ContextualTabGroupHeader="ContextualHeader"
Header="Selected">
    <RibbonGroup>
        <RibbonButton LargeImageSource="Images\HelpFile.ico"
                        Label="Help" />
    </RibbonGroup>
</RibbonTab>
<Ribbon.ContextualTabGroups>
    <RibbonContextualTabGroup Header="ContextualHeader"
                              Visibility="Visible"
                              Background="Orange" />
</Ribbon.ContextualTabGroups>
```

Here in the preceding design we have created a `ContextualTab`. The `ContextualTabGroupHeader` specifies the unique name that is shown on the title bar of the `RibbonWindow`. It also helps to define `RibbonContextualTabGroup` which needs to be visible through the configuration defined inside the `ContextualTabGroups` property of the ribbon. You can also configure the background of the `ContextualTab`.

There's more...

Ribbon controls supports a lot of features. Let us define some of them.

Special RibbonTooltip for Ribbon-based controls

A Ribbon-based UI element has a special tool tip designed which could be shown to the user when they hover their mouse over a control. Each of the controls from `RibbonWindow` automatically creates a `RibbonTooltip` control when the tool tip properties are specified.

```
<RibbonButton SmallImageSource="Images\ConfigurationSettings.
ico"
              Label="Config"
              ToolTipTitle="Configuration Settings"
              ToolTipDescription="Allows you to change
Configuration settings for the current selection"
              ToolTipImageSource="Images\MyBriefcase.ico"/>
```


In the preceding code, `RibbonButton` has been declared with the following few extra properties:

- ▶ `ToolTipTitle`: Each tool tip shows a title in bold letters
- ▶ `ToolTipImageSource`: This specifies the property which shows the image on the left side of the tooltip
- ▶ `ToolTipDescription`: This shows the description that will be shown on the right side of the tooltip control

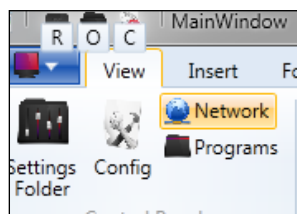
When these properties are specified, the control will automatically create the tool tip when the mouse is hovered over the control for a while.



In the preceding screenshot, **Settings Folder** is the defined title, the image and the description are shown below it. Any control inside the ribbon can show a `RibbonToolTip`.

Specifying shortcut keys for ribbon controls

Ribbon-based controls are smart enough to handle shortcut keys just like the former toolbars and menu bars. Each ribbon control has a special property defined as `KeyTip`, which specifies the key which would focus the control when the `Alt` key is pressed from the keyboard. When the `Alt` key is pressed, the key associated with a particular control is automatically displayed above the ribbon.



In the preceding screenshot, the **R**, **O**, and **C** are displayed when the **Alt** key of the keyboard is pressed.

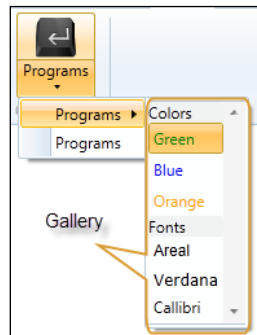
```
<RibbonButton SmallImageSource="Images\RAMDrive.ico" KeyTip="R"
Label="RAM" />
```

When the `KeyTip` is specified, the letter indicates the key which will focus through the `Alt` key, and acts as a hot key for the ribbon.

How to use RibbonGallery inside the ribbon

A `RibbonGallery` is a new set of elements defined in a `RibbonWindow` which maps to a collection. A `RibbonGallery` is a list of the `RibbonGalleryItem` properties that is grouped together using the `RibbonGalleryCategory` properties, which is used to visualize a list in the best possible way that the UI permits. A `RibbonGallery` can be hosted inside controls like:

- ▶ `RibbonMenuButton`
- ▶ `RibbonSplitButton`
- ▶ `RibbonMenuItem`
- ▶ `RibbonSplitMenuItem`
- ▶ `RibbonApplicationMenuItem`
- ▶ `RibbonApplicationSplitMenuItem`
- ▶ `RibbonComboBox`



In the preceding screenshot, we have added a **Menu** in the `RibbonSplitButton`, and a `RibbonMenuItem` inside the `SplitButton` holds the `Gallery`.

```
<RibbonSplitButton LargeImageSource="Images\RUN.ico"
    Label="Programs"
    Click="RibbonButton_Click" >
    <RibbonMenuItem Header="Programs">
        <RibbonGallery SelectedValue="Green"
            SelectedValuePath="Content "
            MaxColumnCount="4">
            <RibbonGalleryCategory Header="Colors">
                <RibbonGalleryItem Content="Green"
                    Foreground="Green" />
            </RibbonGalleryCategory>
        </RibbonGallery>
    </RibbonMenuItem>
</RibbonSplitButton>
```

```
        <RibbonGalleryItem Content="Blue"
Foreground="Blue" />
        <RibbonGalleryItem Content="Orange"
Foreground="Orange" />
    </RibbonGalleryCategory>
    <RibbonGalleryCategory Header="Fonts">
        <RibbonGalleryItem Content="Areal"
FontFamily="Arial" />
        <RibbonGalleryItem Content="Verdana"
FontFamily="Verdana" />
        <RibbonGalleryItem Content="Callibri"
FontFamily="Callibri" />
    </RibbonGalleryCategory>
</RibbonGallery>

</RibbonMenuItem>
</RibbonSplitButton>
```

The preceding code demonstrates the how the galleries are added to the `RibbonMenuItem`. The gallery also supports `ItemTemplate` and `ItemsSource`, which will let you define the source of the gallery from an external list.

Using WeakEvent pattern in WPF

When we are writing a code in the managed environment, it is always a concern how the garbage collector releases the memory footprints and releases for the application. For every application, memory management has been the primary concern. While developing large applications, we often miss looking after any memory leaks that might cause a crash of the application in the long run. If we do take care of our code, we might miss something which is hidden from us due to abstraction made to the language.

Events form special circumstances in a way that the subscriber sends a strong reference to the object to which it is subscribed to, and hence even though the object is not in use later, the subscription to the event is still holding up a strong reference to the object and will remain until the subscription is terminated manually or the main object is disposed. `WeakReference` is not new to the system and is introduced in *Appendix, .NET Languages and its Construct* of the book. `WeakEvent` is a pattern that is introduced to solve this kind of issue.

WPF introduces a new class `WeakEventManager` that bridges the events with the subscriber and maintains only weak reference to the object such that when the object gets disposed, or de-referenced from the user code, the event does not hold any strong reference to it and hence it is exposed to the garbage collector.

How to do it...

In this recipe, we are going to introduce `WeakReferenceManager` and will take you further on how to use this in your real world WPF applications.

1. Start a new WPF project.
2. We create another window and call it `LeakingChild`, and add a reference to the `MainWindow.Activated` event to it, such that when the `MainWindow` of the application gets activated, a special code runs declared within the child window.
3. We can have two approaches to subscribe the `Activated` event from the `ChildWindow`.

```
App.Current.MainWindow.Activated += MainWindow_Activated;
```

The preceding code hooks the `Activated` event of the `MainWindow`, but you must remember that the reference of the object of the `LeakingChild` class is also passed to `EventHandler`. The delegate associated with the `Activated` event creates a list and stores the reference of the `LeakingChild` object. Even if the user closes the `LeakingChild` window, the memory allocated by the object does not get erased. We can detect this by creating a large chunk of memory inside `LeakingChild`.

4. WPF provides `WeakEventManager`, which allows you to subscribe an event to an event handler without having strong reference to the subscriber, such that when the window gets closed, the memory is exposed automatically for garbage collector.

```
WeakEventManager<Window, EventArgs>.AddHandler(App.Current.MainWindow, "Activated", MainWindow_Activated);
```

5. We can replace the previous code with the new code to have a `WeakEvent` implementation. The `MainWindow_Activated` event handler still gets called, but without holding any strong reference to the leaking object.

We detect the memory allocation by explicitly calling `GC.Collect` from the user code and monitoring `PrivateMemorySize64` of the process memory.

How it works...

`WeakEvent` is an implementation of `WeakReference`. As I have already introduced `WeakReference` before in the previous chapter, it uses a `WeakReference` object to hold the target passed for the `EventHandler`. `WeakReference` ensures that the object is always exposed to garbage collection and hence when the object is not in use, the `GC.Collect` can detect it and automatically collect the memory.

Moreover, generally events do not de-reference themselves automatically. When you subscribe to an event, the `WeakEvent` pattern starts listening to the event. So when the event is raised, it calls the handler that is passed to it and stored in `WeakReference` only if the object exists in context. When the object goes out of the context and collected by `GC.Collect`, the `WeakEvent` pattern automatically unsubscribes the event.

It is best to use `WeakEventManager` to overcome memory leaks for an application.

See also

- Read more about `WeakEventManager` at:
<http://bit.ly/WPFWeakEventManager>

6

Building Touch-sensitive Device Applications in Windows 8

The goal of this chapter is to help you understand Windows store applications and also get you familiar with some of the interesting considerations that you need to maintain while building your touch-sensitive applications in Windows 8:

- ▶ Building your first Windows 8 style tiles application using JavaScript, HTML5, and CSS
- ▶ Writing a library for WinJS
- ▶ Building your first Windows 8 style tiles application using C# languages and XAML
- ▶ Working with storage files in Windows 8 style tiles applications
- ▶ Understanding the application life cycle of WinRT applications

Introduction

As an operating system, Windows has gained a considerable amount of popularity and fame around the world. Microsoft as a company is known to the world by far for Windows rather than anything else. The era of Windows has seen a lot of glory and appreciation as it is one of the largest selling general purpose software that runs in the market. Windows is designed to run on any hardware (or rather most of the hardware) even those which are not released as yet. People prefer Windows because of its flexibility of vendors and integration with other software and by far with its user-friendly look and feel. But with the evolving market and technology, there were a large number of tablets and devices that were flooded into the market, each having a unique behavior and facilities which are not totally supported by the Windows operating system.

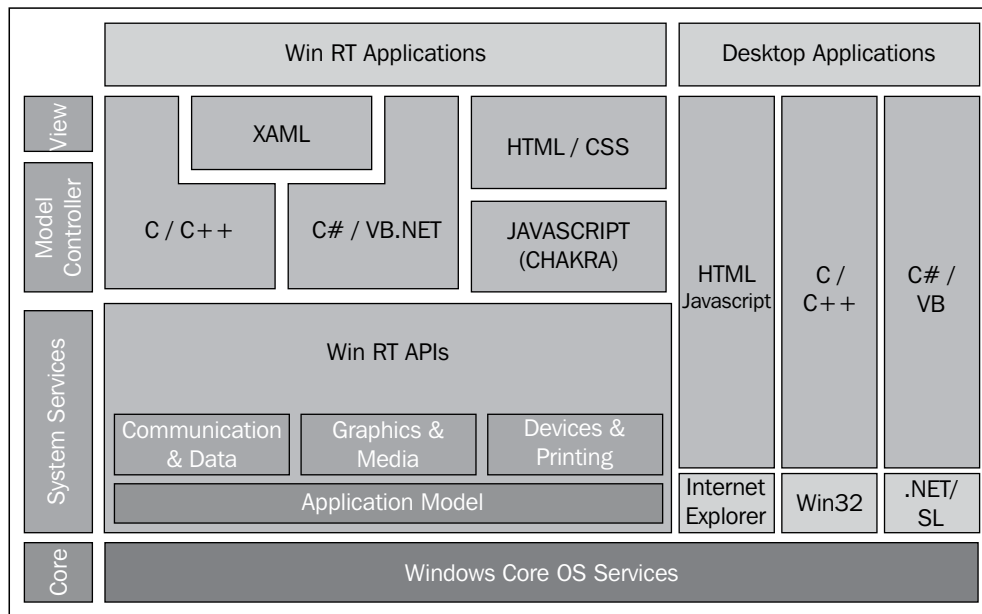
There was a need for a new operating system that can handle these kinds of devices and also take help from the evolving technology. To provide the best end user experience to these devices, Microsoft has to raise the bar to provide specific software that have capabilities to run on these devices. Lately, Microsoft has come up with their new Windows that has inherent support of touch and sensors. We call the new version of Windows, Windows 8.

There are two versions of Windows 8 that came into the market. The one that works on the desktop called Windows 8 Pro, and another that works on arm processors known as WinRT. Microsoft has confirmed that they had redesigned the whole new Windows environment starting from the chip level to the level of abstraction. Therefore, the whole application model in comparison to the developer side has been changed. Even though the core components still remain like the COM environment, a new mode has been developed on top of the core API to support the new style of application development. A new runtime has been built that supports HTML5 and JavaScript to build desktop applications or even other Microsoft-centric languages have evolved new avenues for developers. The introduction to a generalized marketplace for the Windows environment has also paved the way for developers to reach out to the world to a greater extent.

The world of WinRT supports a varied number of languages to do the same job. The job of the languages is to communicate with the hardware devices using COM objects. It should be noted that the COM APIs that are present within the WinRT environment are not actually what we see in traditional COM components. WinRT provides a higher level of abstraction that eliminates the call to the general P/Invoke style programming, rather than placing a subscriber/publisher model using .NET-inspired delegates and events. It should also be noted that WinRT objects do not implement the `IDispatch` interface.



WinRT objects implement `IUnknown` and `ref` counting. They are unmanaged COM APIs exposed in the WinRT environment, which are written in managed languages such as C# to map each type. For instance, the managed CLR type `IEnumerable` is mapped with the unmanaged `IIterable<T>` interface of WinRT. WinRT even though it is unmanaged, supports metadata (as `.winmd` files), so that it doesn't require P/Invoke to communicate from a managed environment.



The preceding diagram shows how WinRT components are laid out. The internal core of the system is the unmanaged WinRT components. The Application Model is built on top of it that provides an abstraction to the core APIs. The Application Model provides the actual WinRT APIs which in turn call the internal core OS services. The WinRT APIs expose almost all of the APIs available from the Core services and are primarily categorized into Communication & Data, Graphics & Media, and Devices & Printing components. The languages are built on top of the WinRT APIs. A WinRT projection is a way of expressing WinRT in a specific language. For instance, you can build a component in one language and can use the same from another language. The languages that are supported by WinRT are as follows:

- ▶ **C/C++** produces native images after compilations. As WinRT is fully native, the applications developed using C++ do not need CLR/.NET to compile/run WinRT applications. C++/CX are extensions provided by Microsoft that helps in developing WinRT apps.
- ▶ **C#,VB/XAML** produces a sandboxed WinRT application that doesn't have access to all the libraries like File I/O, External Devices etc. The CLR maps the basic types of WinRT components into it and exposes them to the application programmer. The application model supported by .NET languages does not support synchronous versions of a number of methods.

- ▶ WinJS is the most abstract development environment to work on for developing WinRT applications. It is built on top of WinRT components and exposes the functionalities to access HTML5 features and embed into a JavaScript and HTML based application. Even though you can create cool applications using JavaScript and HTML5, WinJS still lacks the support of building reusable components or even it cannot access an external component either.

The section to the right of the figure, depicts the desktop applications. Everything that has been developed earlier will run directly under the desktop tile as normal desktop applications. Unlike Windows 8 style tiles applications, desktop applications are full fledged without any restriction at all and also do not take advantage of the marketplace.

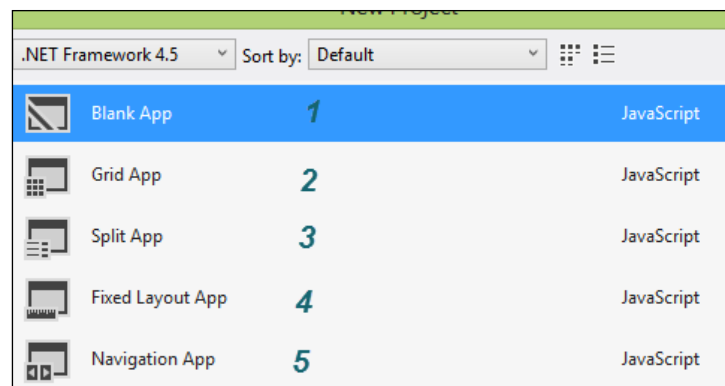
Building your first Windows 8 style tiles application using JavaScript, HTML5, and CSS

If you are coming from a web environment, the new style of desktop application development environment supporting HTML5 and JavaScript to implement applications is your first class choice of development. HTML5 provides superior support of HTML5 to handle WinRT applications leveraging the existing skills of JavaScript and HTML that you already have.

Getting ready

Windows 8 style tiles application has some specific design principles that you need to follow. The layout that we need to follow has already been created as a project template inside Visual Studio 2012. Let us go ahead and create a blank JavaScript application from Visual Studio for the time being.

Visual Studio already provides a number of templates that follow specific rules of the design principles.



In the preceding screenshot, the new project dialog box displays the following project templates:

- ▶ **Blank App:** This is specifically used when you want to build an application from scratch and do not want Visual Studio to help you on building the layout.
- ▶ **Grid App:** This is designed when we need hierarchical navigation. The **Grid App** layout has three-page layers defined. The outermost layer called **Hub pages** displays all the items in a flat grid. Little information about each item is also displayed though. Each of these items are grouped into heads. When an item from the hub page is clicked on, the detail page gets loaded with the details of the item picked by the user. Similarly following the pattern, the **Section page** displays one single group of items when a group head is selected.
- ▶ **Split App:** This is designed to support grouped navigation user interface. The app in this style provides two views. The first page allows group selection and the second page displays each item.
- ▶ **Fixed Layout App:** The application does not scale based on the resolution and maintains a fixed aspect ratio.
- ▶ **Navigation App:** This project template defines navigation buttons already for you.

If we are starting to create the application for the first time, it will prompt you with a dialog box asking for a developer license as shown in the following screenshot. It will ask for your Windows Live ID credentials and finally produce a blank application template.



The blank template gives you a new application with a `default.html` file created for you. Notice, if you see the `css` folder and the `js` folder, you will see a separate `default.js` and `default.css` has already been created as well for you. This pattern is generally followed while creating Windows 8 style applications. For instance, if you are going to add a new HTML page named `home.html`, you will need to add a separate `home.js` file and a `home.css` file as well.

The `default.html` file already reference the JavaScript and CSS file for it. It has also referenced a few JavaScript files such as `base.js` and `ui.js`, which primarily provides the base for creating the application. Another important thing to note is that the `ui-dark.css` file has been added to the HTML that provides the black background. Now let us add some code in `default.html` where it says:

```
<div>
    <label>Enter your Credentials</label>
    <br />
    <label>Enter your userid</label><br />
    <input type="text" id="userId" />
    <br />
    <label>Enter your password</label>
    <br />
    <input type="password" id="userpassword" />
    <br />
    <button id="submitButton">Submit</button>
    <br />
</div>
<span id="outputSpan" />
```

Here we have created normal inputs for `userId` and `userpassword` and a **Submit** button. The code is really simple HTML, and now when you click on **Run** from Visual Studio, you will see the Windows 8 style-based textbox and password box has been replaced by this normal HTML. In the `default.css` file, we add the following CSS:

```
div {
    margin:400px 400px 0 400px;
    border: 1px solid white;
}
```

For simplicity, we placed a border and a margin to the `div` tag that we create on the HTML. Now let us make the HTML a little interactive by adding an event handler for the button using the following code:

```
function loaded() {
    var button = document.getElementById('submitButton');
    button.addEventListener('click', function () {
        var tb = document.getElementById('userId');
```

```

        var pb = document.getElementById('userpassword');
        var span = document.getElementById('outputSpan');

        if (tb.value == pb.value)
            span.innerHTML = 'password is correct';
        else
            span.innerHTML = 'password is incorrect';
    });
}
app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.launch) {
        loaded();
    }
};

```

In the preceding code we call the `loaded()` function when the application is launched. `args.detail.kind` will be set to `launch` during the launch of the HTML. This is similar to `onReady` of a web page. The `loaded` method adds an event handler to the button click event and provides a value for the span. If you run the project, you will see the application will now respond to the button click.

Now having said that, it is not only the HTML controls that we can use and work with when working with Windows 8 style tiles applications. There are special complex controls that are specific to the WinJS as well. Let us look into some of the interesting controls that can be used in your Windows 8 style tiles application in this recipe.

How to do it...

1. Start the application that we are already working on and add a new page to it, we will call it `landingPage.html`. From the **New Item** dialog box, select **PageControl** to add a page with its JavaScript and CSS files. We create a new folder called `html` and place the HTML file inside it. We also place the `css` and `js` files in their respective folders.
2. We cut the HTML that we defined inside the **default.html** page and place it inside content section of the `landingPage.html`.
3. Create a `div` tag in the **default.html** page. We set the `div` tag's `id` element to `hostDiv`. We also place a `data` attribute to the `body` element to declare the path of our landing page:

```

<body data-landingPage="/html/landingPage.html">
    <div id="hostDiv" ></div>
</body>

```

The declarative syntax is generally used by Windows 8 style tiles application to clearly define the navigation.

The idea of placing the `hostDiv` tag is to load the content HTML fragments on the fly based on the user input. It is important to define navigation within the application such that different pages can be opened based on user interaction.

4. We port the JavaScript that we placed in `default.js` as well, and placed the same in `landingpage.js`. Let's place the loaded function inside the `ready` method defined in `landingPage`.
5. Once the JavaScript is ported correctly, we create our `loaded()` function in `default`. As `default.html` acts as a shell that hosts all the other inner pages, we need to implement navigation from this page. We add the following code:

```
function loaded() {  
    homepage = document.body.getAttribute('data-landingPage');  
    WinJS.Navigation.navigate(homepage);  
}
```

The preceding code gets the attribute of `data-landingPage` and navigates the page to this location. `WinJS.Navigation` handles all the navigation inside the window (for instance, it maintains back, forward, and so on). The `navigate` method invokes the `navigated` event to load the location passed in.

6. We handle the `navigated` event that occurs when the `navigate` method is called to set the Promise to load the location inside `contentHost`:

```
WinJS.Navigation.addEventListener('navigated', navigated);  
function navigated(e) {  
    var url = e.detail.location;  
    var host = document.getElementById('hostDiv');  
    if (host.winControl && host.winControl.unload)  
        host.winControl.unload();  
    WinJS.Utilities.empty(host);  
    e.detail.setPromise(WinJS.UI.Pages.render(url, host, e.detail.  
state).then(function () {  
        WinJS.Application.sessionState.lastUrl = url;  
    }));  
}
```

The first line adds an event listener to the `navigated` event. The `navigated` event is called when the `navigate` method is called and the page is rendered and loaded into the host. `e.detail.location` gets the location of the page that has been passed. Here we first release the content of the host div using `WinJS.Utilities.empty` and then we call `setPromise` to asynchronously render the content of the URL directly inside the host. `then` is a method that creates the continuation task for Promise, where the application `lasturl` is set, so that we can use the `WinJS.Navigation` APIs to get back or forward to links. We also called the `unload` event of the existing page before emptying the content of host div.

7. Now this code is fully reusable and when you run the application, it automatically calls `landingPage`.
8. When we call the `navigate` method with the URL, it will automatically call the `navigated` event and load the URL inside `div`. It is always a better idea to load a page inside the content host such that we don't move out of the shell. We create another page and call it `about.html`. We place an anchor tag inside the `default.html` file and point to the page as shown in the following code:

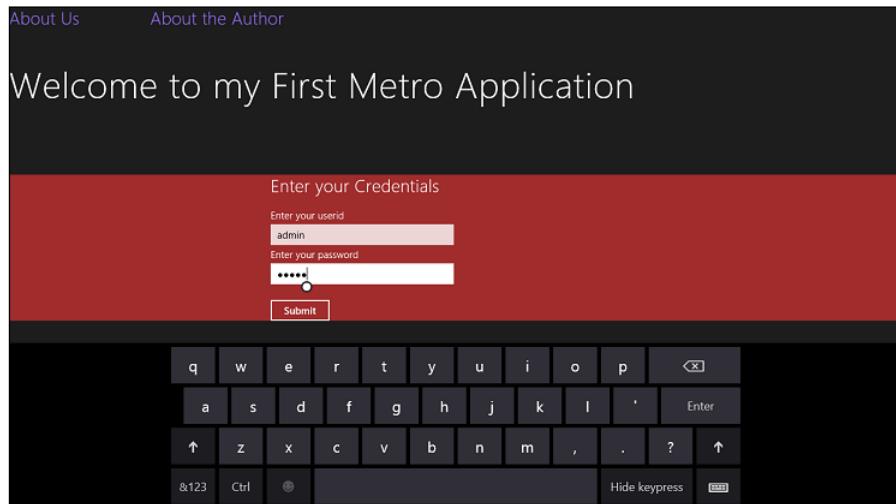
```
<a href="/html/about.html" >About Us</a>
<a href="http://about.me/abhisheksur" target="MyIFrame"
style="margin-left:100px;">About the Author</a>
<iframe name="MyIFrame" style="width:100%;height:100%"></iframe>
```

9. In the preceding code we have placed two anchors, one of which points to a local page while the other points to an external page. To navigate to an external page, we need an `iFrame`. The `MyIFrame` `iFrame` automatically loads the content from an external source whenever the external link is clicked.
10. On the contrary, when the internal link is clicked, it loads the whole page inside the main window and the whole page gets navigated. To load the page inside the host placed in the `default.html` page, we need to listen to the click on an anchor and replace the default navigation to the call to `navigate` as shown in the following code:

```
WinJS.Utilities.query('a').listen('click', processAnchorClick);
function processAnchorClick(e) {
    if (!e.target.href.indexOf("http://") == -1) {
        e.preventDefault();
        var link = e.target;
        WinJS.Navigation.navigate(link.href);
    }
}
```

The first line listens to every click on an anchor tag and calls the `processAnchorClick` method. So whenever any anchor is clicked on in the document, `processAnchorClick` will be called automatically. We check whether this is an internal or an external link. To have the external link to be loaded into `iFrame` we need to bypass this call.

11. The `preventDefault` method prevents the default navigation of the document to the page, but rather we call the `navigate` method manually to load the content in `hostDiv`.



The preceding screenshot shows what the application looks like when loaded. You can see the **About Us** and **About the Author** links on the top of the screen, each of which navigates to load the internal and external document to the page.

12. Let's add a new page called `Input.html` and navigate the same when user authentication is successful. We change the code to validate the `Submit` button click using the following code:

```
if (tb.value == pb.value)
    WinJS.Navigation.navigate("/html/input.html");
else
    span.innerHTML = 'password is incorrect';
```

In this code we navigate to the `input.html` page when the password is correct.

13. In Windows 8 style based applications, even though we have all the HTML5 controls, there are some other special WinJS controls as well, which we can use in our application. Let us add a few of them in the `input.html` file:

```
<div id="createAppBar" data-win-control="WinJS.UI.AppBar" data-
win-options="">
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-opt
ions="{id:'cmdSubmit',label:'Submit',icon:'accept',section:'global
',tooltip:'Submit item'}">
        </button>
```

```

    <hr data-win-control="WinJS.UI.AppBarCommand" data-win-options
    ="{type:'separator',section:'global'}" />
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-opt
    ions="{id:'cmdCancel',label:'Cancel',icon:'cancel',section:'global
    ',tooltip:'Cancel item'}">
        </button>
    <button data-win-control="WinJS.UI.AppBarCommand" data-win-opt
    ions="{id:'cmdCamera',label:'Camera',icon:'camera',section:'select
    ion',tooltip:'Take a picture'}">
        </button>
</div>

```

Each of the WinJS controls is placed inside the code using `div`, and the control is mentioned using the `data-win-control` attribute. `WinJS.UI.AppBar` provides the application bar for the application. The application bar is a special area, which is shown to the user when they swipe the bottom or top of the screen. You can choose `data-win-options="{placement : 'top'}"` to place appbar on the top of the window. Each `data-win-options` control takes a JSON object for the properties of the corresponding WinJS control. The `AppBarCommand` buttons define the ID, label, icon, and so on. We can add event listeners to these buttons just like regular buttons:



The app bar comes to the window on demand and loads the buttons defined inside it. The global section of the app bar is placed from the right-hand side, and generally remains intact while the selection section starts from the left-hand side of the screen. We can use an `hr` tag to create a separation between two `AppBarCommand` buttons.

14. The Appbar commands are defined in the `Input.js` file. We add an event listener for when the submit or cancel button is clicked on to show a message box:

```

ready: function (element, options) {
    var submitBtn = document.getElementById("cmdSubmit");
    submitBtn.addEventListener("click", doClickSubmit,
false);

    var cancelBtn = document.getElementById("cmdCancel")
    cancelBtn.addEventListener("click", doClickRemove,
false);
}

```


The `doClickSubmit` and `doClickRemove` listeners pops up a message box to show the content using the following code:

```
function showMessageBox(msg, title) {
    var msgbox = new Windows.UI.Popups.MessageDialog(msg, title);
    msgbox.commands.append(new Windows.UI.Popups.UICommand("OK",
function () { }));
    msgbox.showAsync();
}
function doClickSubmit() {
    showMessageBox("Thank you for submitting your survey. We will
get back to you soon", "Thank you");
}
function doClickRemove() {
    showMessageBox("You have cancelled the survey. Please come
back soon.", "We are sorry!");
}
```

The preceding code opens a message box with the message and title and with one `UICommand` value `OK`. The `Windows.UI.Popups.MessageDialog` class automatically creates a pop up message box for the user. The message box is modal to the window.

15. Now let us define the main content for the window:

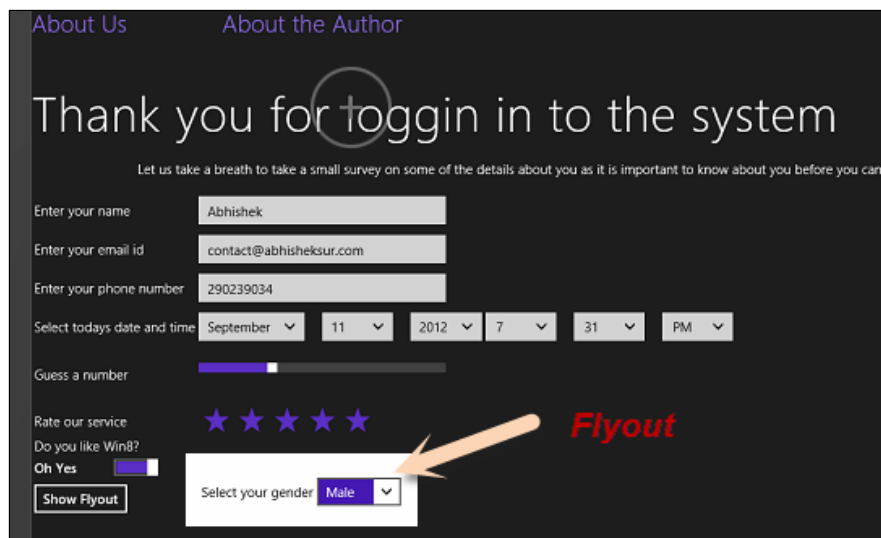
```
<label>Enter your name</label>
<input type="text" /><br />
<label>Enter your email id</label>
<input type="email" /><br />
<label>Enter your phone number</label>
<input type="tel" /><br />
<label>Select todays date and time</label>
<div data-win-control="WinJS.UI.DatePicker" data-win-
options="{maxYear : '2015', minYear:'1990'}"></div>
<div data-win-control="WinJS.UI.TimePicker"></div>
<br />
<label>Guess a number</label>
<input type="range" /><br />
<div id="flyout" data-win-control="WinJS.UI.Flyout">
    <label>Select your gender</label>
    <select>
        <option value="male">Male</option>
        <option value="female" selected="selected">Female</option>
    </select>
</div>
<label>Rate our service</label>
```

```

<div data-win-control="WinJS.UI.Rating"></div>
<br />
<div data-win-control="WinJS.UI.ToggleSwitch" data-win-
options="{labelOn:'Oh Yes', labelOff : 'Not at all', title : 'Do
you like Win8?'}"></div><div data-win-control="WinJS.UI.Tooltip"
data-win-options="{ innerHTML:'A special <b>button</b>!'}">
    <button>Check toolTip</button>
</div>

```

16. Here we first define a few HTML controls to accept the name, e-mail address, phone number, range of numbers, and so on. `WinJS.UI.DatePicker` and `WinJS.UI.TimePicker` define the date and time picker controls for the window.



The preceding screenshot depicts the final UI for the HTML. The HTML input controls behaves properly by providing proper keyboard to the user.

17. The `WinJS.UI.Flyout` control creates a small message box, which flies out either when something is selected or clicked outside the screen. To open a flyout we use the following code:

```

btnFlyout.addEventListener('click', function () {
var flyout = document.getElementById('flyout');
    flyout.winControl.show(btnFlyout);
});

```

Here the `flyout` value is the ID of the `div` tag where the flyout is loaded. When the button is clicked on, we call the `show` method to show the flyout panel adjacent to the control passed in. Here the flyout loads adjacent to the `btnFlyout` button.

18. There is a special `RatingControl` variable that allows you to rate something on the UI. We can handle the events of the controls in the same way as we did for other controls. For instance, say we want to show a message box when the user rates using `RatingControl`. To do this we add an `id` value of `RatingControl`, we call it `rtcontrol`. We can then add an event listener to the `change` event of the control and handle the event when rating is done:

```
var ratingcontrol = document.getElementById('rtcontrol').
winControl;
ratingcontrol.addEventListener('change', function (e) {
    showMessageBox('you have rated ' + ratingcontrol.userRating,
    'Thank you for rating');
});
```

You will see when using WinJS control that you need to call the `winControl` property of the DOM element to get the actual WinJS control. The `change` event defined inside the WinJS control can then be handled normally.

`ToggleSwitch` allows you to select a Boolean value. The options define the title and the on and off captions.

WinJS allows you to produce very cool tooltips using the `WinJs.UI.Tooltip` control. This control allows you to place an HTML tooltip above a control.

How it works...

Each Windows 8 styled application supports a number of languages. Some of the languages are close to the WinRT APIs such as C++, while some are most abstract ones such as WinJS. The HTML5 and JavaScript support is one of the interesting additions to the Windows 8 style tile application development. The HTML5 controls are redesigned totally for Windows 8 style applications using WinJS libraries. There are a number of JavaScript library files associated with the project automatically that form the entire framework of WinJS. With each of the JavaScript library files, there is an associated metadata XML file associated. The metadata defines the pluggability of the files with external libraries. Currently, you cannot define a control using a WinJS library, but you can leverage HTML5 constructs easily for developing cool applications.

Let us discuss the files that are useful in terms of where the WinJS library is concerned:

- ▶ `base.js`: This defines namespaces of all the library classes and adds it to the WinJS namespace. The `base.js` files form the entire library and define its construction. Generally all the JavaScript classes are defined with `strict` turned on.
- ▶ `ui.js`: This defines all the WinJS as well as HTML controls and all the associated methods and properties associated with them. This class is also responsible for generating any UI-related libraries such as animation, behaviors, and many more.
- ▶ `windows`: This maps to the external devices and APIs related to them. The API dynamically creates itself during runtime based on the capabilities of the application.

In this recipe, I have given you a brief introduction on how to create a basic WinRT application using WinJS. Even though the library is well built, you must have already noted that HTML is not capable of creating the UI for WinJS. Specifically, the WinJS controls need to call the `process` method to create the actual WinJS object from `div` that we place for a control. The call to `process` automatically detects the actual control that it needs to construct and calls the constructor of the actual JavaScript library class that is associated with the control defined in `data-win-control` and also find the options available from the JSON object passed to the `data-win-option` attribute defined declaratively into `div`. Once the object has been constructed, it can be retrieved from the `wincontrol` property from the DOM object defined.

There's more...

With HTML5 and JavaScript, the application scope and options are endless. We cannot cover these in one single recipe, so here are the things that have been missed out.

How to deal with the look and feel of Windows 8 style tiles application using JavaScript and CSS

Styling an application in a Windows 8 environment is important so that the application suits itself with the user base with customizations. Windows 8 style tiles application uses CSS stylesheets to apply styles to the properties. You can style an application using normal CSS styles to adorn elements of each page.

By default the application also adds a CSS file to every page. The base CSS page allows you to define themes. There are two themes already supported on the box. One is the light theme, which writes content in black over a white background, and the other is the dark theme, which is set by default and writes white content in gray background. The CSS files are `ui-light.css` and `ui-dark.css` respectively.

These base CSS files also define some predefined styles and sometimes it makes a necessity to override the base style.

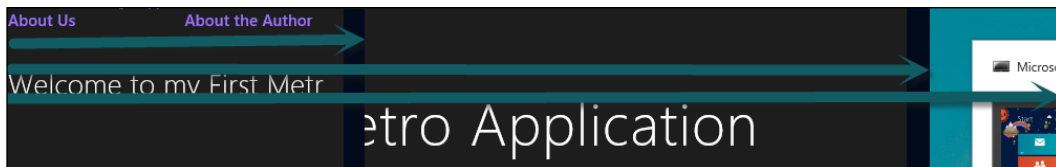
For example, every HTML text control or password control has a button just on the right-hand side of it. You can disable that from your style using the following code:

```
input[type=text]::-ms-clear {
    display:none;
}
```

The cross sign that comes on the textbox is styled using `-ms-clear`. Now to disable it, we can place the preceding code, which overrides the existing style on the textbox and hides the cross from any textbox on the screen. You can look into the `ui-dark.css` stylesheet to get information about any styles applied to the controls.

While styling an application, there are special `@media` tags that you have noticed automatically created for you in the template. The Windows 8 style tiles applications supports three kinds of view:

- ▶ `fullScreen-landscape`: This is the default view of the application where the width of the screen is larger than the height
- ▶ `fullScreen-portrait`: When the device is turned on, it automatically adjusts to the portrait mode where the height is greater than the width
- ▶ `snapped`: The snapped mode takes one-third of the screen where a small portion of the application is snapped with another app
- ▶ `filled`: In case of a filled mode, the app takes two-thirds of the screen



The style `@media` elements are defined for each of these states of the screen. When the window is snapped, you can specify unique styles such that it looks good on the small section of the screen. Similarly, you can specify unique styles for the `filled` view, which takes two-thirds of the screen and finally for the full screen:

```
body {  
    background-color:#808080;  
}  
  
@media screen and (-ms-view-state: fullScreen-landscape) {  
    body {  
        background-color:#ff6a00;  
    }  
}  
  
@media screen and (-ms-view-state: filled) {  
    body {  
        background-color:#00ff90;  
    }  
}  
  
@media screen and (-ms-view-state: snapped) {  
    body {  
        background-color:#0ff;  
    }  
}  
  
@media screen and (-ms-view-state: fullScreen-portrait) {
```

```

    body {
        background-color:#f00;
    }
}

```

In the preceding style, the application changes its background color when the application state gets changed. It gives a different background for portrait, landscape, snapped, filled and full screen views.

How to enable animation within a Windows 8 style tiles application using WinJS

Animation is a special part of any application that creates a professional look and feel to the user and also make them more engaged to the application. Most users interact with the application, and animation provides a smooth finish on their feel and makes it very attractive to them. WinJS has an built-in library that allows the pages, fragments, or individual user interfaces to animate itself in response to the user's activity. The `WinJS.UI.Animation` namespace exposes a lot of cool functionalities that can help animate UI elements quickly and easily. Let us take a look at how to animate objects. We use the existing application that has been created in the recipe to add the animation functionality:

```

app.onloaded = function (args) {
    var host = document.getElementById('hostDiv');
    var anim = WinJS.UI.Animation.enterPage(host, { top: '20px', left:
'300px' });
    anim.then(null);
}

```

In the preceding code, an `enterPage` animation is used which creates a transition animation when the host is loaded. `enterPage` takes two arguments, the first one being the element to animate and the second one being the `top` and `left` offset to animate. You can pass any element to animate the DOM when loaded. You can pass `document.body` to animate the whole document at a time as well.



For simplicity we have used the `onloaded` event to animate the DOM. Generally, it should be written inside the `navigated` event to make it animate whenever page is navigated.

Similarly, you can also have the fade animation using `fadeIn` and `fadeOut` calls.

Let us do it inside the `navigated` event call:

```

WinJS.UI.Animation.fadeOut(host).then(function () {
    WinJS.Utilities.empty(host);
    e.detail.setPromise(WinJS.UI.Pages.render(url, host,
e.detail.state).then(function () {

```

```
        WinJS.Application.sessionState.lastUrl = url;
    }));
    WinJS.UI.Animation.fadeIn(host);
});
```

In the preceding code, during the transition, `hostDiv` which loads the UI gets faded out first using the call `fadeOut`. As any animation automatically uses Promises, you can call them? to set the content and finally show `div` again using `fadeIn`. If you notice, we have added the call to `fadeIn` inside the `then` method right after all content is loaded. This is important as `fadeOut` is an asynchronous call and we need to make sure that `fadeIn` is called after the `fadeOut` is completed. This creates a very good transition effect to the user.

There are also very cool animations called `pointerDown` and `pointerUp` animations, which gives life to elements on the UI. Even though `pointerDown` and `pointerUp` are just animations, using it will make the user interaction very responsive, as they will see that the UI has responded to their call.

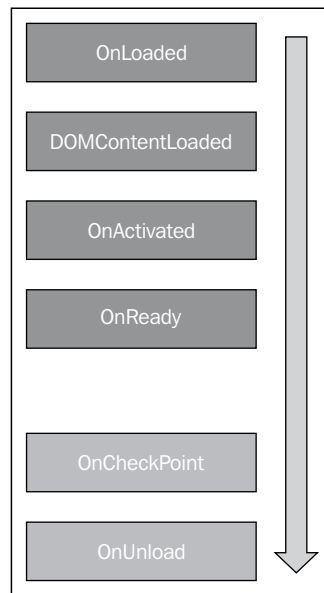
Let us add a `pointerDown` animation to all the titles of the pages. We can query clicks of all the `span` elements from default page such that we can run the `pointerDown` animation effect when the user clicks on the title of any page:

```
WinJS.Utilities.query('span').listen('click', function (e) {
    WinJS.UI.Animation.pointerDown(e.target).done(function () {
        WinJS.UI.Animation.pointerUp(e.target);
    });
});
```

The preceding code listens to the `click` event of any `span` and calls the function that has been passed. The function then runs the `pointerDown` animation on the target and also invokes the `pointerUp` animation when it finishes. The `pointerDown` animation will make the clicks feel like responding with a small change in size which is already optimized by the environment.

How does the event life cycle work in WinJS

There are a number of events that fire when the application gets loaded for the first time. The events that are generated when the application is first loaded are depicted in the following diagram:



During the loading of the application, the `OnLoaded` method is called first, followed by `DOMContentLoaded`, then `OnActivated` and finally `OnReady`. During the application life cycle, there might be times when the application gets suspended or terminated, in such situations, the application generates the `OnCheckPoint` and `OnUnload` events that are the points to save the state of the application before the application gets terminated.

The initial events are important to identify when we fetch the controls. Generally, the controls get loaded before the `OnLoaded` method is fired, and hence the application can call any control on that.

Remember, even though you do not handle these events, the application gets loaded with normal HTML controls, the WinJS controls will not load until an explicit call to process the control is called.

During the initial load up, the application needs to call:

```
WinJS.UI.processAll()
```

This call will process all the WinJS controls or otherwise you can also call `WinJS.UI.Process(control)` to process individual control during the initial application load up.

Writing a library for WinJS

WinJS allows organizing your JavaScript object by giving you an API just like the prototype available in browsers. The WinJS library itself is built using the same API. In this recipe, let us try to build our own library and use it later on, so that we can create our own API for reusability in different sections of the same or separate applications. We will define how you can create your own type using WinJS namespace and classes. We will also discuss the API that helps in defining a class, a namespace, deriving a class, and adding a mixin.

How to do it...

1. The `WinJS.Class.define` API is used to define a JavaScript type that can be used later. The API allows you to supply a constructor function and a set of instance members. You can also pass static members to a type using the same API:

```
var MyType = WinJS.Class.define(function (name) {
    //constructor
    this.name = name;
    this.myInstanceProperty = 30;
},
{
    //JSON to define member property and methods
    myInstanceProperty: 0,
    myInstanceMethod: function () {
        return "from instance";
    }
},
{
    //JSON to define static property and methods
    MyStaticProperty: true,
    MyStaticMethod: function () {
        return MyType.MyStaticProperty;
    }
});

var myobj = new MyType("hi");
myobj.name = "mynewname";
myobj.myInstanceProperty = 20;
myobj.myInstanceMethod();

MyType.MyStaticProperty = false;
MyType.MyStaticMethod();
```

The preceding code calls `WinJS.Class.define` to define a type. The API takes three parameters, the first one being the method that needs to be called as a constructor, and the second being the JSON object as a parameter. The first JSON it uses for instance members and second JSON parameter is used for static members.

The `myobj` object is created from the type `MyType`, which has defined the properties `name` (created inside the constructor), `myInstanceProperty`, and a method `myInstanceMethod`. The static members `MyStaticProperty` and `MyStaticMethod` are directly available on the type `MyType`.



Visual Studio is smart enough to automatically create the IntelliSense property based on the type for your help.

2. WinJS also allows you to define a property using the `Object.defineProperty` API of WinJS:

```
Object.defineProperty(MyType.prototype, "myNewProperty", {
    get: function () { return "" }
});
var value = myobj.myNewProperty;
```

Here we have added a new instance property to `MyType`. The `MyType.prototype` instance adds `myNewProperty` to the instance, while if you need to add as static, you can pass `MyType` itself.

3. The `WinJS.Class.derive` API is used to derive an existing type. This is the same as the inheritance of a type. Let us see the following code:

```
var MyDerivedType = WinJS.Class.derive(MyType, function (name) {
},
{
    //Instance Members
},
{
    //Static members
});
var mydobj = new MyDerivedType("hello");
myobj.myInstanceProperty = true;
```

The code derives an existing type and adds new properties or methods to it. You can also override a method inside it by redefining the same again. As per OOP rules, static members are not derived automatically.

4. Just like what we see in the .NET environment, the WinJS also allows you to wrap your types inside a namespace using the `WinJS.Namespace.define` API:

```
WinJS.Namespace.define("SpecialTypes", {
    MyType: WinJS.Class.define(function (name) {
        //constructor
        this.name = name;
        this.myInstanceProperty = 30;
    },
    {
        //JSON to define member property and methods
        myInstanceProperty: 0,
        myInstanceMethod: function () {
            return "from instance";
        }
    },
    {
        //JSON to define static property and methods
        MyStaticProperty: true,
        MyStaticMethod: function () {
            return MyType.MyStaticProperty;
        }
    }
));
var myobj = new SpecialTypes.MyType("hi");
```

Here `MyType` is wrapped inside the `SpecialTypes` namespace, such that when we try to create an object of the type, we need to specify the namespace in front of the type.

5. Mixin are special objects that implement certain functionalities. WinJS allows you to pass a JSON object and mix the same with an existing type such that all the members defined within that type are automatically copied to the type. The `WinJS.Class.mix` namespace is used to mix a type with another:

```
var mixin = {
    myMethod: function () {
        return "myMethod";
    }
};
WinJS.Class.mix(SpecialTypes.MyType, mixin);
var mixobj = new SpecialTypes.MyType("helloworld");
mixobj.myMethod();
```

So here the `mixin` object has a method called `myMethod` that has been mixed to the type called `SpecialTypes.MyType`, which adds it to the type. Mix is generally used to copy functionalities defined within an external library inside its own type without defining it again.

WinJS JavaScript APIs allow the developer to get the full flavor of object-oriented principles without going through the existing complexity of the JavaScript API.

See also

- You can refer to the following link:

<http://bit.ly/Win8Javascript>

Building your first Windows 8 style tiles application using C# and XAML

XAML is the local .NET language that is also supported while building a Windows 8 style tiles application. Just like HTML5, XAML is an XML-based design language that has the inherent capability of working with .NET languages and is built on top of .NET. XAML is important to define the UI while the code is written either in C# or VB.NET or any .NET language supported by WinRT. In this recipe we are going to use XAML and C# to work with our Windows 8 style applications.

Getting ready

Let us start the recipe by creating a blank XAML based Windows 8 style application. When the application is open, you will see a file called `App.xaml` opened for you. This file provides the main application level configurations that you can apply on the application. For instance, if you want to define a common resource, or override search or sharing charm behaviors or even changing the startup page, this is the right place to look at.

To start with the application let's start creating a UI for login:

```
<Popup x:Name="lgCtrl" IsOpen="False">
    <StackPanel Orientation="Vertical" Background="Red"
x:Name="pop" >
        <TextBlock Text="Enter your credentials"
HorizontalAlignment="Center" Foreground="White" FontSize="25" />
        <TextBlock Text="Id" Margin="10" Foreground="White"
FontSize="25" />
        <TextBox x:Name="txtid" />
        <TextBlock Text="Password" Foreground="White"
FontSize="25" />
```

```
        <PasswordBox x:Name="txtpwd" Height="40"
Margin="5,1" Width="408" />
        <Button x:Name="btnLogin" Click="btnLogin_Click_1"
Foreground="Wheat" Width="100" Content="Submit" ></Button>
        <TextBlock x:Name="tbStatus" Foreground="White" />
    </StackPanel>
</Popup>
```

If you have read *Chapter 7, Communication and Sharing using Windows 8*, already, you must be already familiar with how to work with XAML. Here I have defined a simple XAML control that uses a pop up to load the UI onto the screen. Pop ups load the screen over some screen in the background. It invokes the `btnLogin_Click_1` event handler when the button is clicked.

When the button is clicked, we either show some message depending on the validity of the password, or navigate to another screen called `input.xaml`. To do this we use the following code:

```
private void btnLogin_Click_1(object sender, RoutedEventArgs e)
{
    if (!txtid.Text.Equals(txtpwd.Password))
        tbStatus.Text = "Password is incorrect";
    else
        this.Frame.Navigate(typeof(input));
}
```

The `Frame` object holds the reference of the outer navigation frame. When a page is navigated to using the `Navigate` method, the page automatically loads the page in the UI. Now let us explore some of the controls that can be used in Windows 8 style tiles applications using XAML in this recipe.

How to do it...

1. Start Visual Studio 2012 and open `input.xaml` and write some code to explore some of the other controls that useful in XAML:

```
<StackPanel Orientation="Horizontal">
    <HyperlinkButton Content="About Us" Click="HyperlinkButton_
Click_1" />
    <HyperlinkButton Content="About the Author"
Click="HyperlinkButton_Click_2" />
</StackPanel>
<TextBlock Text="Thank you for logging in to the system"
Style="{StaticResource PageHeaderText}" />
```

Here we have created two `HyperLinkButton` objects that place a link on the UI. These buttons need the `Click` handler to work with.

2. There are a lot of styles defined for any XAML-based project. If you open `Common\StandardStyles.xaml`, you will see a number of styles predefined within the project that can be used later on when required. In the preceding XAML, the style called `PageHeaderTextStyle` has been defined inside `StandardStyles.xaml`.
3. We add a new page and named it `About Us`. We add some text to this page such that the page doesn't look empty. The first `HyperLinkButton` object will navigate the UI to this page and the second link will get HTML from the server and load it to the UI.

4. Let us add a `webView` control which is used to navigate to an HTTP link:

```
<WebView x:Name="webView1" Visibility="Collapsed" />
```

We have made it invisible initially and show only when the page is navigated to an external site.

5. Let us add content to handle the button clicks as shown in the following code:

```
private void HyperlinkButton_Click_1(object sender,
RoutedEventArgs e)
{
    this.Frame.Navigate(typeof(AboutUs));
}

private void HyperlinkButton_Click_2(object sender,
RoutedEventArgs e)
{
    Uri targetUri = new Uri(@"http://www.about.me/abhisheksur");
    webView1.Navigate(targetUri);
    webView1.Visibility = Windows.UI.Xaml.Visibility.Visible;
    spMain.Visibility = Windows.UI.Xaml.Visibility.Collapsed;
}
```

In the preceding code, the first button just navigates to a page created on the inside of the project. The second link navigates to an external site and make the `WebView` control visible. Hence, when the second link is clicked, the HTML content from the Web is shown over the `WebView` control.

6. Most of the applications define an application bar to list the common commands. Windows 8 style tile applications support the application bar to be added either on the top of the screen or on the bottom of the screen. Let us define the application bar for our page using the following code:

```
<Page.BottomAppBar>
    <AppBar x:Name="bottomAppBar" Padding="10,0,10,0">
        <Grid>
            <StackPanel Orientation="Horizontal"
HorizontalAlignment="Left">
```

```
        <Button Style="{StaticResource
PhotoAppBarButtonStyle}" Click="Button_Click"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal"
HorizontalAlignment="Right">
        <Button Style="{StaticResource
SaveAppBarButtonStyle}" Click="Button_Click"/>
        <Button Style="{StaticResource
DiscardAppBarButtonStyle}" Click="Button_Click"/>
    </StackPanel>
</Grid>
</AppBar>
</Page.BottomAppBar>
```

We have added an AppBar control on the BottomAppBar content of the page. This enables the application bar to kick in when the user swipes from the bottom of the screen. The AppBar control holds buttons inside it, which has been styled by some of the predefined styles in the standardStyles resource file.

7. Now let us look into the click handler of the buttons of AppBar:

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    var msgbox = new MessageDialog("You have clicked " + (sender
as Button).Name, "AppBar button clicked");
    var uiCommand = new UICommand("Thanks");
    msgbox.Commands.Add(uiCommand);
    await msgbox.ShowAsync();
}
```

The preceding code creates a message box and shows the same on the screen with one button to cancel the message box with the label "Thanks" on it. A message box is a small modal dialog which displays message to the user.

8. Now let us write the main content of the application:

```
<TextBlock Text="Enter your name" />
<TextBox Grid.Column="1"/>
<TextBlock Text="Enter your email id" Grid.Row="1" />
<TextBox InputScope="EmailSmtAddress" Grid.Row="1" Grid.
Column="1"/>
<TextBlock Text="Enter your phone number" Grid.Row="2" />
<TextBox InputScope="TelephoneNumber" Grid.Row="2" Grid.
Column="1"/>
```

In the preceding code, we have introduced `InputScope` for a control. The `InputScope` property defines the input type of a textbox. Windows 8 automatically sees the `InputScope` property and provides the appropriate on screen keyboard to the user. You can see that in addition to a normal textbox, you can also define an e-mail textbox, a telephone textbox, and so on.

9. A slider gives the user an option to specify a value from a range of numbers. The **Guess a number** field defines a `Slider` control that shows a range of numbers:

```
<TextBlock Text="Guess a number" Grid.Row="3" />
<Slider Grid.Row="3" Grid.Column="1" />
```

10. `ToggleSwitch` is a unique control to toggle between two states. The `ToggleSwitch` control has `OnContent` and `OffContent` properties, which are shown when the switch is on or off. `ToggleSwitch` is similar to `ToggleButton` but visually they are different:

```
<TextBlock Text="Are you sure you like this app?" Grid.Row="4"/>
<ToggleSwitch OnContent="Yes" OffContent="No" Header="Liked it ?"
Grid.Row="4" Grid.Column="1"/>
```

11. `ToggleButton` is bound with a pop up, which is a flyout panel displaying some content to the user when the button is clicked:

```
<ToggleButton Content="Show Popup" Grid.Row="5"
x:Name="tbtnOpenPopup"/>
<Popup IsOpen="{Binding ElementName=tbtnOpenPopup,
Path=IsChecked}" FlowDirection="LeftToRight"
IsLightDismissEnabled="True">
    <StackPanel Orientation="Vertical" Background="White">
        <TextBlock Text="Enter your gender" Foreground="Black" />
        <ComboBox>
            <ComboBoxItem Content="Male" />
            <ComboBoxItem Content="Female" />
        </ComboBox>
    </StackPanel>
</Popup>
```

Here the pop up is bound to the `tbtnOpenPopup` control on its `IsChecked` property. When `ToggleButton` is pressed, it opens the pop up automatically as it invokes a change on the `IsChecked` property, to show the content inside it.

How it works...

Windows 8 supports designing applications using XAML and C#. XAML is a powerful technique to define the UI and bind the UI with the object code. The WinRT defines APIs to support XAML-based programming techniques. In this recipe, we have covered some of the interesting controls that are present in XAML environments and can be used while creating an app in XAML technique. The XAML window contains a few sections:

- ▶ **Frame:** It defines the shell of the program. You can load user controls inside one shell page, or you can use the navigation UI to change the frame's main page.
- ▶ **StandardStyles:** Inside the common folder of each project, there are a number of styles defined automatically by the project templates. The common styles are useful to define unique user experiences for an app.
- ▶ **App:** Each application provides an `App` class that configures the application. The basic usage of the `App` class is to provide the starting point of the application.

Each XAML file has a corresponding `.cs` file associated with it. The CS file creates a `.NET` source file which can be coupled with the XAML design to create interactivity. The WinRT types are converted to `.NET` interfaces and are exposed using C# `.NET` languages. The reference to `.NET` Windows 8 style applications has automatically taken which refers to all the types that are present in `.NET` library and also the types that maps to the WinRT COM types.

There's more...

After creating one of the most basic UIs using Windows 8 style tiles applications let us consider some of the other interesting additions that makes sense to know.

What are the layouts available for a Windows 8 style tiles applications

Windows 8 style tiles application supports a number of layouts as we have already seen in the JavaScript section of this chapter. The supported layouts are as follows:

- ▶ **Snapped**
- ▶ **Filled**
- ▶ **FullScreen**
- ▶ **Portrait**
- ▶ **Landscape**



It is also worth noting that successful implementation of these layouts is also required for the app to be submitted to the Windows store.

Snapping makes the app 320px wide and allows the app to be placed beside another app. When one app is snapped with another, two-thirds of a portion of the screen is taken by another app while the rest is taken by this app. The `VisualStateManager` class is used to display various states of the app.

While designing an app, it should be taken in mind to create an app to support all the snapping modes. Let us define how we can adjust the layout in Windows 8 style tiles applications:

```
<VisualStateManager.VisualStateGroups>
    <!-- Visual states reflect the application's view state
-->
    <VisualStateGroup>
        <VisualState x:Name="FullScreenLandscape">
            <Storyboard>
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Filled">
            <Storyboard>
            </Storyboard>
        </VisualState>

        <VisualState x:Name="FullScreenPortrait">
            <Storyboard>
            </Storyboard>
        </VisualState>

        <VisualState x:Name="Snapped">
            <Storyboard>
            </Storyboard>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Even though we are creating a clear free flowing UI that can be adjusted easily in all types of resolutions, there is `VisualStateManager.VisualStateGroups` which can be named as `Snapped`, `FullScreenPortrait`, `Filled`, or `FullScreenLandscape` to define the animation which needs to be performed when `VisualState` changes from one state to another. The `VisualStateManager` class can be written on any screen and indicates how the page will respond to various layout events.

Generally inside `VisualStateManager` we hide some portion of the visual and show some other portion depending on the size of the screen:

```
<VisualState x:Name="Snapped">
    <Storyboard>
```

```
<ObjectAnimationUsingKeyFrames Storyboard.  
TargetName="itemListView" Storyboard.TargetProperty="Visibility">  
    <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>  
</ObjectAnimationUsingKeyFrames>  
<ObjectAnimationUsingKeyFrames Storyboard.  
TargetName="itemGridView" Storyboard.TargetProperty="Visibility">  
    <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>  
</ObjectAnimationUsingKeyFrames>  
</Storyboard>  
</VisualState>
```

For instance, in the preceding XAML, when the page is snapped, `itemListView` is made visible which is created for 320 px and `itemGridView` is for all other visual states. The `ObjectAnimationUsingKeyFrames` class defines the animation of objects by defining every keyframe.



Application snapping is only available to devices that are running with a resolution of more than 1366px.

How to implement animation in Windows 8 style tile applications

In Windows 8 style tiles application, animations play a very vital role in terms of user experiences. From the opening of Charms to slide animation of the settings or even animation inside the application itself, all make up the experiences that the application user gets. Windows 8 applications has built-in support for animation, for instance controls such as `ToggleSwitch`, `AppBar`, `GridView`, and `ProgressRing`. All have in-built animation and they all animate on either, with the user interaction or without it.

Although you can build your own custom animations using `Storyboards` as we have already discussed in *Chapter 7, Communication and Sharing using Windows 8*, there are a lot of in-built animation libraries already present for you, which can be used into your application to maintain really consistent behaviors to the user. All the library animations are already optimized and you can take advantage of using them on the go.

In Windows 8 style tiles application we have either the Theme Transitions or Theme Animations.

- **Theme Transitions:** These animations are automatically invoked when they are added to certain state changes. For example, `EntranceThemeTransition`, `RepositionThemeTransition`, `AddDeleteThemeTransition`, `ReorderThemeTransition`, and so on.

- **Theme Animations:** These animations need to be explicitly invoked by the application developer. For example, `FadeInThemeAnimation`, `FadeOutThemeAnimation`, `PopInThemeAnimation`, `PopoutThemeAnimation`, `SplitOpenThemeAnimation`, `SplitCloseThemeAnimation`, `TapUpThemeAnimation`, `TapDownThemeAnimation`, `RepositionThemeAnimation`, and so on.

Let us look at how to define the Transitions and Animations in code:

```
<Grid Style="{StaticResource LayoutRootStyle}">
  <Grid.ChildrenTransitions>
    <TransitionCollection>
      <EntranceThemeTransition FromHorizontalOffset="500" />
    </TransitionCollection>
  </Grid.ChildrenTransitions>
</Grid>
```

In the preceding code, a `EntranceThemeTransition` object is defined for the Grid transition, such that when the grid is loaded for the first time, the UI elements will move from a horizontal offset of 500px. This will give a smooth animation behavior to the user.

Theme Animation on the other hand is explicitly called. To define a Theme Animation, we need to define a storyboard inside a `VisualState` class which will be triggered automatically when the control goes into that state. The animation can also be invoked by handling custom events on the control, but it is recommended to use `VisualStateManager` instead:

```
<VisualState x:Name="Opened">
  <Storyboard>
    <SplitOpenThemeAnimation OpenedTargetName="Border"
                             ContentTargetName="ScrollViewer"
                             ClosedTargetName="DropDownToggle"
                             ContentTranslationOffset="0" />
  </Storyboard>
</VisualState>
```

Here when the control fires the `Opened` event, `SplitOpenThemeAnimation` is triggered and the animation gets executed. We can also use our own custom animation inside the storyboard.

See also

Visit the following link for more examples:

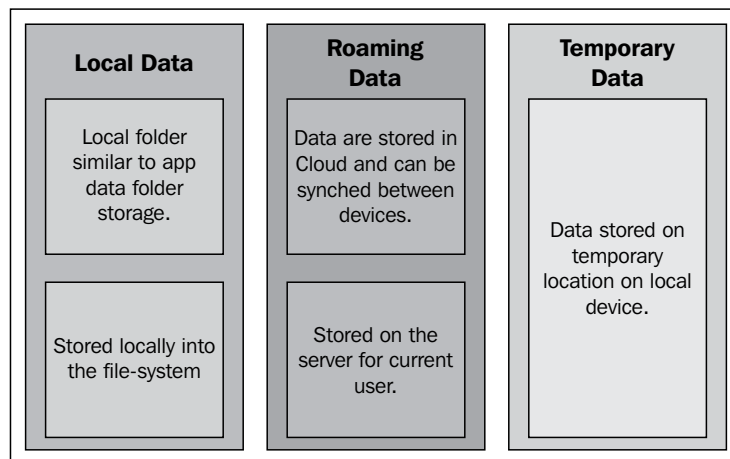
<http://bit.ly/Windows8XAML>

Working with storage files in Windows 8 style tiles applications

Data persistence is one of the important concerns of any device running any application. While working on any app, the first thing that the user might want is to persist some data inside the app such that when the user comes back, the data remains intact even after several days. Even though files can be used on the devices with the support of specifying local system drive locations, this is not the recommended way of defining persistent storage. We can use declarations to access the pictures library, music library, and so on, but the data will not be app-specific or would not belong to any security that the app can take advantage of. The local storage on these storage locations are often not very secure as the user has direct access to these locations.

Windows 8 style tiles application provides the API to support storage in the application itself. They provides both local storage and roaming profile storage such that the application can share or sync data across multiple devices. The app data can be user isolated, such that the data associated with one user cannot be accessed by another user even on the same device. The usage of storage classes while storing data inside the app is recommended.

There are in fact three kinds of storage:



The preceding figure shows how the three kinds of storages are laid out. The app data files and settings are laid out to local data folders.

The data is set as private to the device. When using the roaming profile, the data is synched with the cloud such that the data is available to multiple devices. The temporary data storage are local temporary storage files.

How to do it...

In these steps, we are going to cover how to work with storage in Windows 8 style tiles-based application:

1. Start a new application and let's call it `LocalDataStorageSample`. Open the project and create a file. We call it `AppDataManager`.
2. Write two methods to store and retrieve data from a local file using the `StorageFile` API:

```
public static async Task SaveToLocalDataAsync<T>(T data, string
filename)
{
    StorageFile file = await ApplicationData.Current.
LocalFolder.CreateFileAsync(filename, CreationCollisionOption.
ReplaceExisting);
    var stream = await file.OpenStreamForWriteAsync();
    var serializer = new DataContractSerializer(typeof(T));
    serializer.WriteObject(stream, data);
    await stream.FlushAsync();
}
```

This code creates a file on the Local folder of the application. The Local folder is the local `AppData` folder which the application is associated with. It is an isolated storage that the application has access to. The code saves the data we pass into the method into the filename. The steps are to open `OutputStream` and write serialized data into the file. `FlushAsync` actually flushes the buffered data into the file.

3. Now let us create a method to retrieve the information stored into the file. Let us write the following code:

```
public static async Task<T> RestoreToLocalDataAsync<T>(string
filename)
{
    try
    {
        var file = await ApplicationData.Current.LocalFolder.
GetFileAsync(filename);
        var stream = await file.OpenStreamForReadAsync();
```

```

        var serializer = new DataContractSerializer(typeof(T));
        return (T)serializer.ReadObject(stream);
    }
    catch { return default(T); }
}

```

4. The preceding code is just the reverse to the previous one. Here `StorageFile` is opened and it uses `InputStream` to deserialize the data stored into the file and a string is created and returned back.
5. Create a textbox (named `txtData`) on the UI and two buttons to load and save the data. We create button handlers to save and load data respectively:

```

string filename = "localData.txt";
private async void Button_Click_1(object sender, RoutedEventArgs e)
{
    await AppDataManager.SaveToLocalDataAsync<string>(txtData.
Text, filename);
}

private async void Button_Click_2(object sender, RoutedEventArgs e)
{
    txtData.Text = await AppDataManager.RestoreToLocalDataAsync<st
ring>(filename);
}

```

The preceding code creates a file named `localData.txt` on the Local folder when **Save** is clicked. The data from the file is retrieved in the same way as using the **Load** button.

6. Similar to `LocalFolder`, the current application can also hold `ApplicationSettings` which will be stored into the registry. Let us create some helper methods to store and retrieve settings:

```

public void SaveSetting<T>(string settingName, T value)
{
    ApplicationData.Current.LocalSettings.Values[settingName] =
value;
}
public T GetSetting<T>(string settingName)
{
    try
    {
        var localsettings = ApplicationData.Current.LocalSettings.
Values;
    }
    catch { }
}

```

```

        if (localsettings.ContainsKey(settingName))
        {
            return (T)localsettings[settingName];
        }
    }
    catch { }
    return default(T);
}

```

The above code sets the `LocalSetting` value based on the `settingName` we pass to it and save the value to the `settingName`. The `GetSetting` returns back the object that has been previously stored into settings.

7. Rather than using `LocalFolder` or `LocalSettings` always, the `StorageFile` API provides a `RoamingFolder` or `RoamingSettings` data folder as well, such that when the application saves data, it directly stores it to their roaming profiles and when another device opens the same file, it gets the data from the roaming folder. `RoamingFolder` points to the user's SkyDrive.

We can use `ApplicationData.Current.RoamingFolder` instead of `ApplicationData.Current.LocalFolder` to access the roaming folder for the current user.

8. Application data settings supports containers for composite data storage. Let us look into the following code:

```

public void SaveSetting<T>(string settingName, T value, string
containerName, bool roam = false)
{
    var dataSettings = roam ? ApplicationData.Current.
RoamingSettings : ApplicationData.Current.LocalSettings;
    var container = dataSettings.CreateContainer(containerName,
ApplicationDataCreateDisposition.Always);
    container.Values[settingName] = value;
}

```

In the preceding code, the `dataSettings` variable creates a new container using `CreateContainer`. It is a catalog folder where a set of values can be stored. So for each container there is a collection of `Values` objects where we are saving our settings.

9. While using `RoamingData`, there are times where the `AppData` gets changed to another device and the current sync operation of the roaming profile gets the changed data to the local device. In such scenarios, you can use the `ApplicationData.Current.DataChanged` event that gets called when data has been changed on the current folders.

How it works...

Storage classes give special access to filesystem folders. From a Windows 8 style tile application, a local filesystem is not available. Hence to access a local filesystem, the only way is to use the `LocalStorage` classes, even though the application can invoke the capability to store in some special known folders. The superior API to store into the local storage is capable of storing files and settings, where the files are stored into the `AppData` folder of the current users application and the settings are stored into the local registry.

The `LocalStorage` class also supports the usage of the `RoamingData` folders where the application stores the files into the SkyDrive associated with its current profile.

`RoamingData` is used to synch data between devices such that the user can have a unique user experience by using the application. These data automatically syncs up with the SkyDrive in a bandwidth-friendly way by a background system service that runs automatically to do this.

There's more...

Let us consider a few more options that we can explore regarding the recipe.

How to work with the Settings charm for an application

Every application provides some settings for the application. Windows 8 style tiles application provides a charm button that automatically opens up a sliding panel to apply settings for a particular application that the user is running. Even though there are some basic settings for every application which are displayed on the panel, the application programmer can also add custom settings to this pane by directly accessing it from the application. Let us look into how to work with it:

```
public MainPage()
{
    this.InitializeComponent();
    SettingsPane.GetForCurrentView().CommandsRequested += MainPage_
CommandsRequested;
}

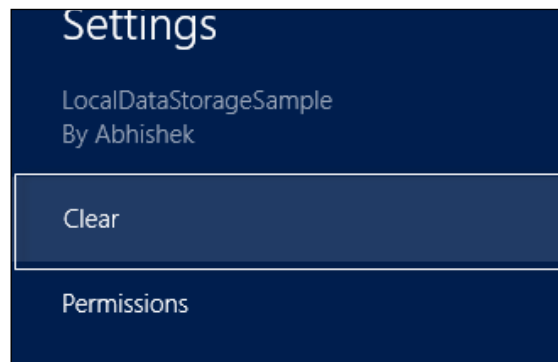
void MainPage_CommandsRequested(SettingsPane sender,
SettingsPaneCommandsRequestedEventArgs args)
{
    var clearTextBoxSettings = new SettingsCommand("Clear Data",
"Clear",
        cmd =>
        {
```

```

        this.txtData.Text = string.Empty;
    });
    args.Request.ApplicationCommands.Add(clearTextBoxSettings);
}

```

In the preceding code we handled the `CommandsRequested` event of `CurrentView` of `SettingsPane`. This event is triggered when the application user opens the **Settings** charm. In the event handler we have placed a **Clear** command to clear data on the screen. Adding `SettingsCommand` to the `ApplicationCommands` collection will make the **Settings** button be displayed on the **Settings** pane that comes when the **Settings** charm is clicked:



The preceding screenshot depicts how the settings charm displays the custom clear button with the default permission button that exists for every application. The **Clear** button will invoke the user code defined inside the `Application` command.

How to take pictures from the camera in a Windows 8 device

Windows 8 style devices contains a web camera. Now from a Windows 8 style tile application, it is easy to get a picture from the webcam. Let us look at how to get an image from the webcam.

To do this, first you need to add a declaration on the `Package.appxmanifest` file to have access to the webcam. Once you have access, you can write the following code to get an image from the camera:

```

public class CameraCapture
{
    public async void TakeSnapshot(string filename)
    {
        var ui = new CameraCaptureUI();
        ui.PhotoSettings.CroppedAspectRatio = new Size(4, 3);
        StorageFile file = await ui.CaptureFileAsync(CameraCaptureUIM
ode.Photo);
    }
}

```

```
        if (file != null)
        {
            var stream = await file.OpenAsync(FileAccessMode.Read);
            StorageFolder storageFolder = KnownFolders.
PicturesLibrary;
            var result = await file.CopyAsync(storageFolder,
filename);
        }
    }
}
```

This code takes a picture from the camera and writes it to `PicturesLibrary`. To access the pictures library, the application also needs to add a capability on the manifest.

The `CameraCaptureUI` class gets a file from the `CaptureFileAsync` call. When the photo is captured, we crop the image in a 4:3 aspect ratio and save it to the pictures library.

See also

- ▶ The *How to share application data between applications using Windows 8 environment* recipe in *Chapter 7, Communication and Sharing using Windows 8*
- ▶ Refer to the following link:

<http://bit.ly/Windows8Storage>

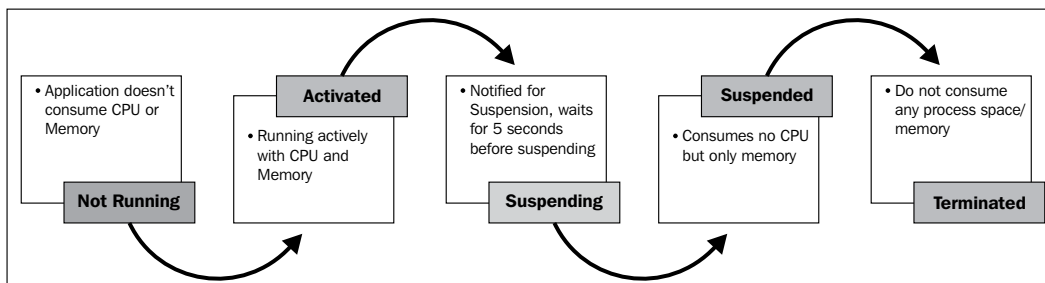
Understanding the application life cycle of WinRT applications

Windows 8 applications are quite different than the normal desktop applications. The applications that need to build on this environment have to undergo certain changes, or you need to take care of certain special significances while building an application. For example, in case of normal Windows applications, the user has the close buttons for each application which is used to close the application, while in Windows 8 style tiles application, the user does not have a close button in general (and it is explicitly recommended not to have one). The user has to switch between different apps rather than closing one and opening another. Another important fact is that the Windows operating system core also takes care of the memory usage on the box or the availability of memory, based on which it might suspend an existing app or terminate it. Thus even though the application hasn't been stopped by the user explicitly, the process can be terminated automatically by the operating system. Again, Windows 8 style tiles app are sandboxed and have a lot of restrictions to it. It cannot do anything to the computer which users are allowed to do. The application code is treated as far less trusted in the Windows 8 style tiles application.

The desktop applications provides full knowledge to the user on what state the program is in. The application doesn't close itself, even when there are memory crunch scenarios. Generally in traditional applications, the application that has been executed later will theoretically have less performance than the applications that run before. Each application will have a secure button on the taskbar. But in the case of Windows 8 style tiles application, things are quite different. The application makes sure that the latest application will have the same application performance than those launched earlier. The applications will not hold the taskbar but rather produce a special ribbon on the left-hand side of the screen to display recently launched applications. The applications also won't run when they are not in the foreground. When the application is placed in the background, it goes into the suspended mode and doesn't consume CPU memory after a while. It will still go on using the memory it is allotted for, but the operating system can terminate the application when it sees a memory crunch. This system has the following three benefits:

- ▶ Multiple applications to run on the same machine without any memory crunch situation
- ▶ It reduces the burden to shutdown the application by hand
- ▶ It prevents the application in the background from consuming CPU and memory in the background, and hence the process running in foreground will have full advantage

But to have a better experience from the application, the user shouldn't be aware of what happened to the application. The application should be developed in such a way that the user, when switching between applications, will have an impression that the it has been running all along. So as a developer, it is a little bit harder to improve end users' experiences:



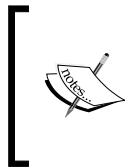
In the preceding diagram, the application is first in the Not Running state where the application is not running. The user activates the application using the Start menu and uses it actively. Now at a certain time, the user can switch to another application. Windows will then suspend the application. It will first wait for 5 seconds to ensure the application is not reactivated. It then pushes it into the Suspending mode, where the user is notified for the last time to save the state. It gives 5 seconds more to save the application state or any non-trivial information and finally it goes into the Suspended mode.

In Suspended mode, the application is not allowed any CPU usage but memory still remains there. The operating system terminates the program only when the application is not used for a long time and there is a memory crunch.

In this recipe we are going to cover how to work with various application life cycle stages.

Getting ready

In the case of using C#-based Windows 8 application projects, we are provided with the `App` class which generally provides all the application level events. The `App` class invokes various special events that can be used to handle the application life cycle stages. Let us start a blank project from Visual C# Windows 8 style tiles application and name it.



For JavaScript-based projects, the Shell page contains events such as `WinJS.Application.onactivated` or `WinJS.Application.oncheckpoint` to do similar tasks. The `onactivated` event is equivalent to the `OnLaunched` event and `oncheckpoint` is equivalent to `OnSuspending` as discussed here.

In the `app.xaml.cs` page, you will see a number of events already defined for you. This comes with the project template. There are two events handled in this class. One is `OnLaunched`, which indicates whether the application is started by the user. This may hold the state of the previous run. The other events are `OnSuspending` and `OnResuming`. The `OnSuspending` event is helpful when the application is being suspended and gives you 5 seconds of time to save the data and state. The `OnResuming` event is called whenever the application is turned to activated from the Suspended state. Generally, we need to save data during this time and also the state, where the application is in before the application gets suspended. This will ensure that if the application gets terminated and gets activated again, we can restore the state and data that the user has already been working on.

The idea of the API is to create a simple UI that will store the data when modified on the app on the fly. I mean the application will autosave data as it is working on to the local storage folders. When the application gets terminated by the operating system, it will reload the data present on the application during its re-launch. In this way, any temporary data added by the application can be restored when the application gets suspended and eventually terminated.

How to do it...

1. Lets add a model class to the solution. We call it `StateObject`. The `StateObject` class has been made very simple by adding one single property that generates one single GUID. We override the `ToString` method to return the same GUID:

```
public class StateObject
{
```

```

private string _myname;
private string MyName
{
    get
    {
        if (string.IsNullOrEmpty(this._myname))
            this._myname = Guid.NewGuid().ToString();
        return this._myname;
    }
}
public override string ToString()
{
    return this.MyName;
}
}

```

Here is the simple implementation of the Stateobject class.

2. We create another class and call it ModelRepository to add ObservableCollection of a list of StateObject instances. A list box and a button is added and the list box is bound to ObservableCollection:

```

<StackPanel Orientation="Vertical">
    <ListBox ItemsSource="{Binding LocalStates}" />
    <Button Click="Button_Click" Content="Add new item" />
</StackPanel>

```

LocalStates is an observable collection defined inside the ModelRepository class we defined. When Button_Click is fired, we call an Add method of the ModelRepository class.

3. We are going to use StorageFolders to save the state of the app. Let us open the ModelRepository folder to add the following code:

```

private static async Task SaveDataToAsync(StateObject[]
copyToSave)
{
    StorageFolder folder = ApplicationData.Current.LocalFolder;
    StorageFile file = await folder.CreateFileAsync(FILETEMPORARY,
CreationCollisionOption.ReplaceExisting);

    IRandomAccessStream ras = await file.OpenAsync(FileAccessMode.
ReadWrite);
    IOutputStream ostr = ras.GetOutputStreamAt(0);

    var serializer = new DataContractSerializer(typeof(StateObje
ct[]));

```

```
        using (Stream clrStream = ostr.AsStreamForWrite())
        {
            serializer.WriteObject(clrStream, copyToSave);
        }

        await file.RenameAsync(FILEFINAL, NameCollisionOption.
ReplaceExisting);
    }
}
```

This code opens `LocalFolder` of the app and stores an array of `StateObject` to its local folder storage with the filename specified as `FILEFINAL`. Remember, we have used `LocalFolder`. You can also use `RoamingFolder` to store user-specific data or even the cloud to have data stored for a more sophisticated approach.

We use `CreateFileAsync` to create a file if not to exist or replace it. When the file is open, we use `DataContractSerializer` to serialize all the objects we pass to it and write the data to the file created. It is worth noticing that we create a temporary file first, and when the save is done, we rename the temporary file with `FinalFile`.

4. When an object is added to the UI, we need to add the same to the file stored in `LocalFolder`. To do the same, we add a queue message to the `ModelRepository` method which will be called whenever an item is added.
5. From the button click we call the `AddToStateDataAsync` method:

```
public static Task AddToStateDataAsync(StateObject newObject)
{
    ModelRepository.StateObjects.Add(newObject);

    return AddItemToSaveQueue(() => SaveStateData(StateObjects.
ToArray()));
}
```

In this code, in addition to adding the object to `ObservableCollection`, we also add the item to `AddItemToSaveQueue`, where we have passed an array of the entire `ObservableCollection`.

6. We also need to ensure that all the calls to the `SaveQueue` method need to be processed sequentially. To do this, we create a chain of method calls. We use the `async` pattern to await each stack of these calls such that when one is done, we call the other:

```
static Task _stask;
private static Task AddItemToSaveQueue(Func<Task> worksaveasync)
{
    if (_stask == null || _stask.IsCompleted)
        _stask = worksaveasync();
    else
```

```

        _stask = Chain(_stask, worksaveasync);
        return _stask;
    }
    private static async Task Chain(Task current, Func<Task>
    worksaveasync)
    {
        await current;
        await worksaveasync();
    }
    private static Task SaveStateData(StateObject[] copyToSave)
    {
        return Task.Factory.StartNew(() =>
        SaveDataToAsync(copyToSave).Wait());
    }

```

In the preceding code, we maintain the Task object and create a AddItemToSavequeue method. We determine whether the existing task is complete or not. If it is complete, we directly call the function to save data (SaveStatedata) that has been passed to it, or we call Chain to create a chain of await statements.

7. We add a method to await on any existing queue using the following code:

```

public static async Task CompleteAllOutstandingSaveWorkAsync()
{
    if (_stask != null && _stask.IsCompleted)
        await _stask;
}

```

Here, the method only waits for the task to complete.

8. In the OnSuspending event of App.xaml.cs, we call ModelRepository.CompleteAllOutstandingSaveWorkAsync to ensure that all the current save operations are complete before the application gets suspended:

```

private async void OnSuspending(object sender, SuspendingEventArgs
e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    await ModelRepository.CompleteAllOutstandingSaveWorkAsync();
    deferral.Complete();
}

```


It is to be remembered that `await` does not hold the thread. Rather, it will return immediately after calling the method. Thus when the `CompleteAllOutstandingSaveWorkAsync` method is called, it immediately returns the control. `e.SuspendingOperation` exposes a method called `GetDeferral` to notify the environment that the current operation isn't yet complete. We can indicate the complete notification using the `Complete` method of the `SuspendingDeferral` object.

9. As we have saved the existing model data into local storage on the fly, we also need to ensure that the application needs to load the data when it is launched from terminated state. To deal with this we write a method to load data from `StorageFile` created:

```
public static async Task LoadModelAsync()
{
    var io = Task.Factory.StartNew(() => LoadModelAsyncToAsync().
Result);
    IEnumerable<StateObject> storedStates = await io;
    foreach (var obj in storedStates)
        StateObjects.Add(obj);
}

private static async Task<IEnumerable<StateObject>>
LoadModelAsyncToAsync()
{
    StorageFolder localFolder = ApplicationData.Current.
LocalFolder;
    StorageFile file;
    try
    {
        file = await localFolder.GetFileAsync(FILEFINAL);
    }
    catch (FileNotFoundException)
    {
        return Enumerable.Empty<StateObject>();
    }
    IInputStream instream = await file.OpenReadAsync();
    DataContractSerializer serializer = new DataContractSerializer
(typeof(StateObject[]));
    using (Stream clrStream = instream.AsStreamForRead())
    {
        return (StateObject[])serializer.ReadObject(clrStream);
    }
}
```

Here we retrieve the file from `LocalFolder` using the `GetFileAsync` method. Once the file is retrieved, we read the file and deserialize the content using `DataContractSerializer`. Once the objects are returned, we load them to `ObservableCollection` that has been bound to the UI.

10. The `OnLaunched` event of the `App` class gets the `PreviousExecutionState` values. We can get information about whether the application has been terminated or is freshly run. If the application is terminated previously by the operating system, we can load up the data that has been saved to the temporary local storage using the following code:

```
if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
    await ModelRepository.LoadModelAsync();
}
```

We call the `LoadModelAsync` method when the application is launched from termination to ensure that all the unsaved data is restored and the application to the user looks like it's running throughout.

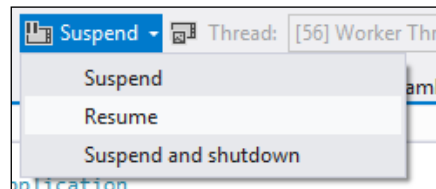
How it works...

Application life cycle for a Windows 8 style tile application generates events. During the initial launch of the application, it first shows the splash screen for at the most 15 seconds and then opens the application. It eventually calls the `OnLaunch` event handler defined for the app to load custom contents to the application. The `Launch` event of the application provides various information about `PreviousExecutionState`, `ActivationKind`, and so on. The `PreviousExecutionState` property defines what happened to the previous execution of the app. If the application is freshly launched, it will give you the `Running` state, while if it is terminated by the operating system, it will give you the `Terminated` value.

The `ActivationKind` property on the other hand provides relevant information about how the application is being launched. The available kinds are as follows:

- ▶ `Launch` (for a normal launch)
- ▶ `Search` (when the application is loaded from search)
- ▶ `ShareTarget` (when the application is launched for sharing)
- ▶ `Device` (when the application is launched on device availability)

There are other options too. To have better user experiences, we need to load the previous state that the user has left behind and load that silently to make an impression to the user that the application has never terminated. In the recipe steps, we have saved the content of the data (probably in your case, you save only the unsaved data) to the local storage folders. The folder will hold a file which stores the XML contents of the data. Now if the application gets terminated by Windows, the application saves the content that it is already autosaving on change during its suspending event. When the application is switched back on, either with resume or after termination, the user state remains intact:



You can try out the app from Visual Studio using the **Suspend/Resume** button on the Debug window or even **Suspend and shutdown** to simulate the terminate operation of the application.

There's more...

An application model can have some very important things to note for Windows 8 style tile applications. Let us look at how to handle different application model states conveniently and efficiently.

Launch of the application and splash screen

Each Windows 8 style tile application is required to have a splash screen. The project templates provided by the Windows 8 style tile application type already provides a default transparent bitmap that show the splash screen when the application is getting launched from the Not Running state. In the `Assets` folder of your project you will find `SplashScreen.png`, which is a transparent image that shows the initial start screen. You can replace the image with your existing image.

The splash screen is of 620 x 300 pixels long. You can open the `Package.appxmanifest` file and scroll down the application UI panel to see the splash screen image listed on the screen:



The preceding screenshot displays the section of splash screen configuration on the `Package.appxmanifest` file. You can also provide a background color for the splash screen such that the color gets through to the transparent background of the bitmap as well.

Remember, the application model restricts the user to show this splash screen for 15 seconds. If the application needs more than 15 seconds to run, you need to create a separate splash screen to mimic the same and run the services in the background.

How to package an application

After building a product, once you are done with the application, the last thing that you need is to package your application and deploy it to your clients. Visual Studio offers you to either deploy through local package files (called **sideloading**) or using Windows developers' account, install through the Web. Though the Windows developers' account is the convenient way of packaging and deploying an application, you need to buy an account to try this.

Let us look at how to package an application for windows store. Open `Package.appxmanifest` and verify the version for the application from the **Packaging** tab. You can also choose a certificate and the **Publisher** name as shown in the following screenshot:

Use this page to set the properties that identify and describe your package when it is deployed.

Package name: 1196f5b2-b449-4fa4-bf85-bfa4de82ef28

Package display name: ApplicationLifecycleSample

Logo: Assets\StoreLogo.png Required size: 50 x 50 pixels

Version: Major: 1 Minor: 0 Build: 0 Revision: 0

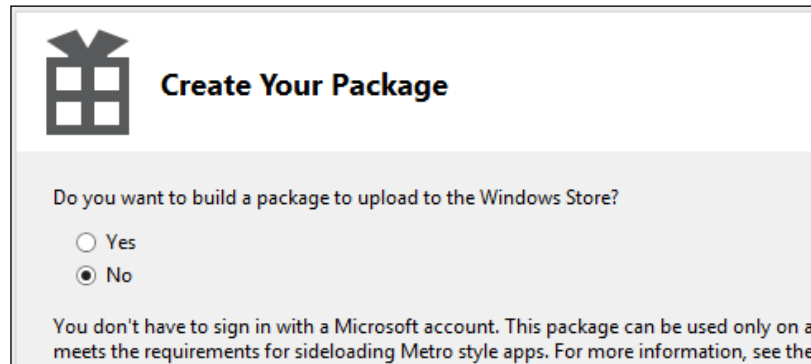
Publisher: CN=Abhishek Choose Certificate...

Publisher display name: Abhishek

Package family name: 1196f5b2-b449-4fa4-bf85-bfa4de82ef28_sq9va4e45x4ta

Here we can see how the application packaging looks. The **Version** field specifies the application version. When creating an update to the same app, you need to increase the version. There is also a **Publisher** name and certificate to choose for the app. The package store logo can also be changed from here.

Once done with the configuration, go to **Project | Store | Create App Package**:



Create Your Package

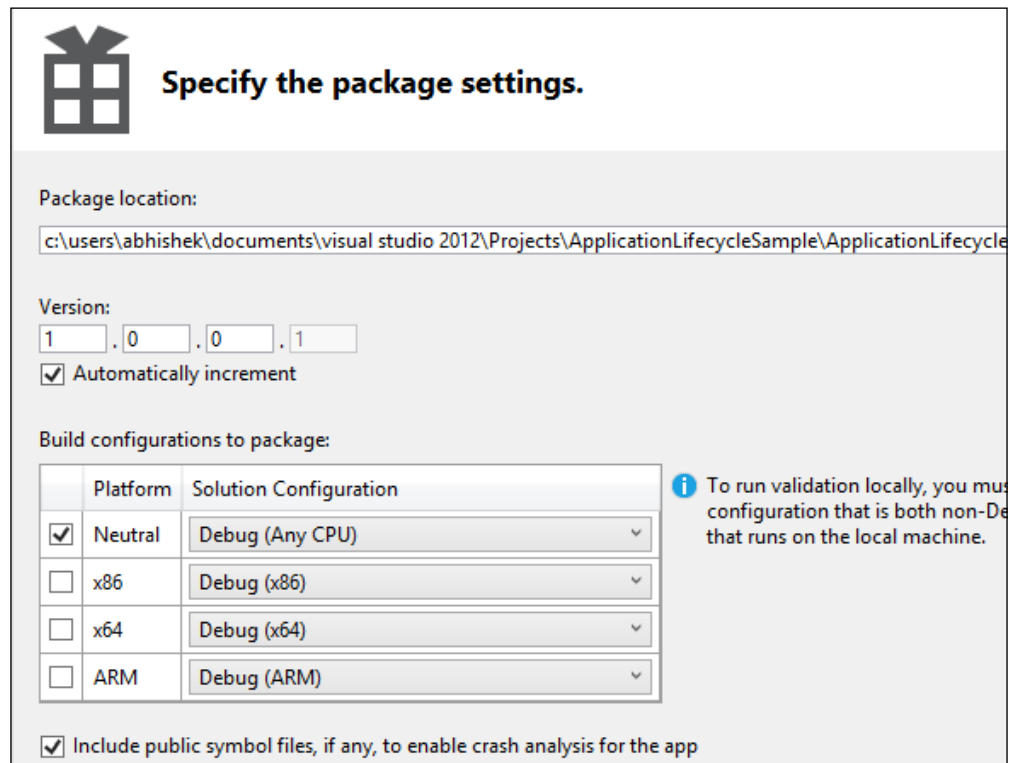
Do you want to build a package to upload to the Windows Store?

☐ Yes

☒ No

You don't have to sign in with a Microsoft account. This package can be used only on a device that meets the requirements for sideloading Metro style apps. For more information, see the Windows Store website.

In the **Create Your Package** dialog, it shows that you can either choose to sign in to your developer account to directly send the package, or just create the file package. Lets, for time being, choose **No** to create the package locally and click on **Next**.



Specify the package settings.

Package location:
c:\users\abhishek\documents\visual studio 2012\Projects\ApplicationLifecycleSample\ApplicationLifecycleSample

Version:
1 . 0 . 0 . 1

☒ Automatically increment

Build configurations to package:

	Platform	Solution Configuration
<input checked="" type="checkbox"/>	Neutral	Debug (Any CPU)
<input type="checkbox"/>	x86	Debug (x86)
<input type="checkbox"/>	x64	Debug (x64)
<input type="checkbox"/>	ARM	Debug (ARM)

☒ Include public symbol files, if any, to enable crash analysis for the app

i To run validation locally, you must specify a configuration that is both non-Debug and that runs on the local machine.

The **Specify the package settings** dialog box shows the location where the package will be created, the version of the file, and also the configuration. The **Any CPU** configuration is set by default, but users can specify target devices of 32 bit or 64 bit or ARM processor architectures. Click on **Create** to create the package.

Once the project is packaged, it will create a `appxupload` file on the location that is specified. It contains the following components:

- ▶ Compiled code and custom DLLs
- ▶ XAML files (not compiled BAML)
- ▶ Resources (binary files, localized resources)
- ▶ Application manifest
- ▶ Digital signatures

The `appxupload` file is generally a packaged file. You can rename the file to a `.zip` extension and unzip its contents to see what is inside. You will find the things stated earlier.

See also

Check out the following links:

- ▶ <http://bit.ly/Windows8AppLC>
- ▶ <http://bit.ly/Windows8AppDeploy>

7

Communication and Sharing using Windows 8

The goal of this chapter is to give you a clear idea on how to deal with communication between applications and devices. This chapter also gives you an insight into how to deal with standard APIs using WinRT libraries. We are going to cover the following topics:

- ▶ How to enable app to app sharing inside a Windows 8 environment
- ▶ Working with notification and services inside the Windows 8 environment
- ▶ How to perform background transfers of data in Windows 8 style tile applications

Introduction

Sharing has been the key element while developing any application in the world. Virtually, no application can be fully functional without the ability to share its resources to the external world. Networking is one of the primary concerns of any application running in any environment. With the introduction of more and more interactivity in technology and more and more adaptation of social media, there is always a rise in the need for APIs to enable easy sharing of resources with the external world. The world of Windows 8 devices has already started to show extraordinary response to the sharing of resources and activities. The operating system has an inbuilt system of sharing resources to the other applications installed on the system, such that the applications that are capable of receiving the resources can be invoked directly to send data. The sharing is not only restricted to the operating system sharing options, but rather there are a large numbers of APIs supported by the Windows 8 environment that make it very easy to share public content and connect to public standard data formats much more easily than we have ever thought.

In this chapter, we are going to see how to use the APIs available with the WinRT environment and create applications that are capable of communicating with the external world and get or send data simultaneously using standard set of protocols available on the market.

How to enable app to app sharing inside a Windows 8 environment

One of the most important and greatest advantages of a Windows 8 style tile-based application is its ability to participate in app to app sharing. By app to app sharing we mean that one app shares content or data with another app based on some predefined standard system contract that the apps need to follow. The application actually interacts with the Windows 8 environment based on the contract defined in it. There are two agents that participate in this app to app sharing. One is the source that indicates the application which invokes the other application and acts as a host of resources, and second is the destination applications which retrieves the data sent from the host in various formats supported by the system and present the data somewhere. A number of social media applications already ship with Windows 8 which might come in handy to make ready transfers of resources directly to social media environments such as Twitter, Facebook, to name a few. In this recipe we are going to take a step-by-step approach to see how application content can be shared between other applications just by creating both the source and destination environments.

How to do it...

The following steps will help enable app to app sharing inside a Windows 8 environment:

1. Start an application and consider creating an application that can act as a sharing source. To handle an app to be a sharing source, we need to handle the `DataTransferManager.DataRequested` event. This event is called when the Sharing charm is opened by the user in the app. We put two textboxes on the screen, which will determine the title and data that need to be sent out. We refer to them as `tbTitle` and `tbData`. Let us write some code to share something from the app:

```
public MainPage()
{
    this.InitializeComponent();
    DataTransferManager.GetForCurrentView().DataRequested +=
MainPage_DataRequested;
}

void MainPage_DataRequested(DataTransferManager sender,
DataRequestedEventArgs args)
{
```

```

var dataPacket = args.Request.Data;
dataPacket.Properties.Title = tbTitle.Text;
dataPacket.Properties.Description = "My custom data sent from
application";

dataPacket.SetText(tbData.Text);
dataPacket.SetUri(new Uri("http://abhisheksur.com"));
dataPacket.SetHtmlFormat(HtmlFormatHelper.
CreateHtmlFormat(string.Format("<b>{0}</b><br/><i>{1}</i>",
tbTitle.Text, tbData.Text)));

var img = RandomAccessStreamReference.CreateFromUri(new
Uri(@"http://abhisheksur.com/370715_1209722123_1468206619_n.
jpg"));
dataPacket.SetBitmap(img);
dataPacket.Properties.Thumbnail = img;
}

```

The `DataRequested` event gets the information on what kind of data the application can share and based on that, the charm will load the share target supporting these data formats. To ensure you get a share charm, it is important to specify more than one data format. Here we have specified the text format, and the HTML format as well. We have set a Bitmap, and depending on the share target the user opens, they can use this information and share the content.

- Share targets on the other hand act as targets for a share from another app. To make an app a share target, we need to add a declaration of **Share Target** to the `Package.appxmanifest` file declaration section:

Available Declarations:
Select one...

Supported Declarations:
Share Target

Description:
Registers the app as a share target, which allows the app to receive shareable content.
Only one instance of this declaration is allowed per app.
[More information](#)

Properties:
Data formats
Specifies the data formats supported by the app; for example: "Text", "URI", "Bitmap", "HTML", "StorageItems", or "RTF". The app will be displayed in the Share charm whenever one of the supported data formats is shared from another app.

Data format	Sharing formats	<input type="button" value="Remove"/>
Data format: Bitmap		<input type="button" value="Remove"/>
Data format: Text		<input type="button" value="Remove"/>
Data format: HTML		<input type="button" value="Remove"/>

3. We also need to define the sharing formats that the application can receive. Let us define Bitmap as the sharing data format to receive Bitmap sent from a sharing source.
4. Now let us add a sharing UI. The sharing UI is different from a normal page, it opens in a shorter screen which takes most of the screen, but yet some portion of the actual app remains open in the background. Let us add a page to our solution. We add some textboxes, images, and some Textblocks elements depending on what you want to share. Next, let us define an Activate method on the page as shown in the following code:

```
private ShareOperation _shareOperation;
public async void Activate(ShareTargetActivatedEventArgs args)
{
    this._shareOperation = args.ShareOperation;

    // Communicate metadata about the shared content through the
    view model
    var shareProperties = this._shareOperation.Data.Properties;
    var thumbnailImage = new BitmapImage();
    this.tbTitle.Text = shareProperties.Title;
    this.tbDescription.Text = shareProperties.Description;
    Window.Current.Content = this;
    Window.Current.Activate();

    // Update the shared content's thumbnail image in the
    background
    if (shareProperties.Thumbnail != null)
    {
        var stream = await shareProperties.Thumbnail.
        OpenReadAsync();
        thumbnailImage.SetSource(stream);
        this.imgThumbnail.Source = thumbnailImage;
    }
}
private void ShareButton_Click(object sender, RoutedEventArgs e)
{
    this._shareOperation.ReportStarted();

    //Share the content
    this._shareOperation.ReportCompleted();
}
```

Here in this code we share the content we receive from the share source. We define a `ShareOperation` object which we receive as argument to the `Activate` method. We set the object properties to our UI and finally use the `ShareButton_Click` method to share the content. You can see in the preceding code that the `ShareOperation` object exposes `ReportStarted` and `ReportCompleted`. These helper methods are used to report to the source app that the sharing has been done, so that the user can close the sharing pop up that has been shown on the screen automatically.

5. We would also need to call the `Activate` method. The `App` class defines an overridable `OnShareTargetActivated` method. When the Sharing charm is invoking the application, this method is automatically called with an appropriate argument. To open our special page, we call `Activate` of the page and load the UI:

```
protected override void OnShareTargetActivated(ShareTargetActivatedEventArgs args)
{
    var shareTargetPage = new Sharing();
    shareTargetPage.Activate(args);
} protected override void OnShareTargetActivated(ShareTargetActivatedEventArgs args)
{
    var shareTargetPage = new Sharing();
    shareTargetPage.Activate(args);
}
```

In the preceding code, we have overridden the `OnShareTargetActivated` method in our code to redirect the call to loading our sharing UI when the application is activated using the Sharing charm. In this way your application can act as a share target or share source or both.

How it works...

Sharing data between one or more applications is supported automatically in the Windows 8 environment. The applications that can share data are called **source applications**, while the applications that are capable of receiving data are called **target applications**. The source application supports a number of formats based on which you need to pass data. The applications that are capable of receiving specific data formats, which is specified by its declarations, are automatically filtered in the environment.

The search also works in the same way as sharing. In the case of making the application work with Search, you need to override the `OnSearchActivated` method in the `App` class. The `OnSearchActivated` method receives `QueryText` as `SearchActivatedEventArgs`, which is the same text that the user types in the Search box.

Working with notification and services

Services form an important part of any application. Most applications need to get data online using services to update content. The applications might also need to update the data through live services. Windows 8 style tile applications support a number of new APIs that are capable of handling network resources easily and elegantly. First of all, to work with network services, the Windows 8 style applications exposes two sets of APIs, which can be invoked easily from inside of the application. They are as follows:

- ▶ General HTTP API
- ▶ Syndication API

The General HTTP API accesses online services through a default HTTP gateway of request and response. It supports basic HTTP services or even RESTful services to handle web content. On the other hand the Syndication API accesses web through standard feeds format. It can also update web feeds using common web standards such as the Atom Feed standard. These APIs are built on top of the WinRT model, which support inherent usage of async services that can be used directly within the app.

The services are the endpoints where the application needs access to get latest content or update content to the Web. To deal with the content, we also need a unified model to enhance users' experiences with notification services such that we can notify the user with new updates automatically. The Windows Store applications expose two types of notifications that can be used inside the app. They are toast-based notifications, which notify the users directly on the screen using a pop up, and then there are tile-based notifications, which update the live time which the user might have pinned to the Start menu.

We can use these services to get content and update the notification to the user even if the application is not running. In this recipe, we are going to cover the basics on how to use the services from our code and how to update the notification to the end user.

Getting ready

To get ready, let us create a RESTful service to handle the CRUD operations on a list of contacts. To do this, create a WCF project and add a `Contact` class as follows:

```
[ServiceContract]
public interface IContactService
{
    [WebGet(UriTemplate = "contact/{roll}",
    ResponseFormat=WebMessageFormat.Json)]
    [OperationContract]
    Contact GetContact(string roll);
}
```

```

        [WebInvoke(Method = "POST", UriTemplate = "contacts",
ResponseFormat = WebMessageFormat.Json)]
        [OperationContract]
        bool SaveContact(Contact currentContact);

        [WebInvoke(Method = "DELETE", UriTemplate = "contact/{roll}",
ResponseFormat = WebMessageFormat.Json)]
        [OperationContract]
        bool RemoveContact(string roll);

        [WebGet(UriTemplate = "contacts", ResponseFormat =
WebMessageFormat.Json)]
        [OperationContract]
        List<Contact> GetAllContacts();
    }
    [DataContract]
    public class Contact
    {
        [DataMember]
        public int Roll { get; set; }

        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public string Address { get; set; }

        [DataMember]
        public int Age { get; set; }
    }

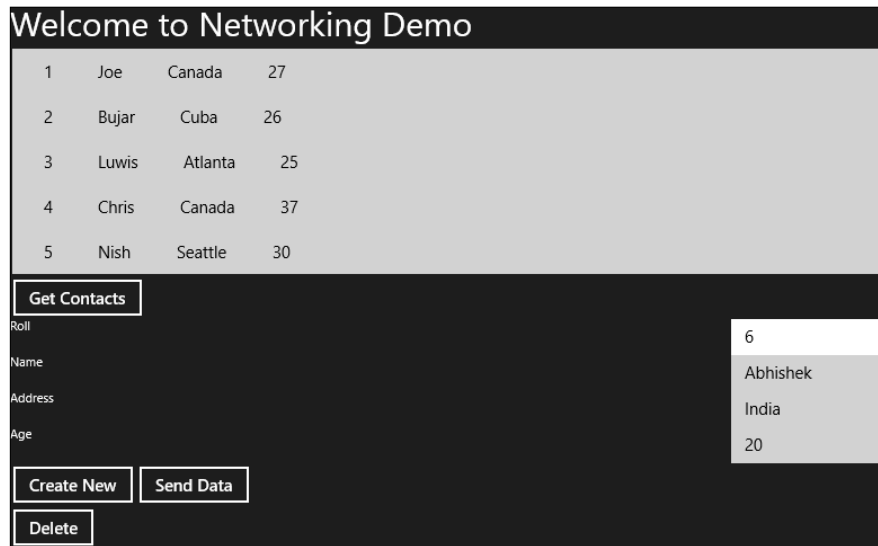
```

Here in this code, we have created a standard WCF service that uses `System.ServiceModel.Web.dll` to create a RESTful service API. Now the `WebGet` method will create a `Get` response service while the `WebInvoke` method is used for any type of verb. The REST service is then hosted on IIS or a small hosting console, and made accessible via a normal HTTP request/response model.

Now let us create a new Windows 8 style application using a blank template and add code to it.

How to do it...

1. On the main page, we add some elements. One button (we call it as `Get Contacts`), a list box to display the data returned from the server, a grid with the properties of each individual item, and two buttons to send data to the server and delete a record. This is illustrated in the following screenshot:



This design has been created using XAML. A `ListBox` control that shows the contacts, a `Button` control to get a contact, a form control to enter a contact, two `Button` controls to send and clear the form, and a `Button` control to delete the form.

2. Create an object of the `HttpClient` type on the constructor and specify the `BaseAddress` value. `BaseAddress` specifies the base location on which the service needs to invoke:

```
client = new HttpClient();  
client.BaseAddress = new Uri("http://localhost:8080/  
contactservice/");
```

You can see our base location is specified, this is where our service is hosted.

3. The `HttpClient` type has few APIs available that can help in getting or sending data to or from a service. The `GetAsync` method gets an `HttpResponseMessage` class, which can be used to retrieve the actual content sent from the service:

```
HttpResponseMessage response = client.GetAsync("contacts");  
response.EnsureSuccessStatusCode();  
string content = await response.Content.ReadAsStringAsync();
```

In the preceding code, the contacts are called on `ServiceLocation`, and the array of contacts is retrieved. The `GetAsync` method gets `HttpResponseMessage`, which holds the entire message that has been received as response. The `EnsureSuccessStatusCode` method generates an exception if anything other than status code 200 is received by the `HttpClient` class. If everything is ok, you can retrieve the content using `ReadAsStringAsync` on the response object. If you see the content object, it will hold the string representation of the actual response.

You can also use `GetStringAsync` to get the string content directly. This is a shortcut method for the previous three lines.

4. To get a list from the item received from `response`, we need to deserialize the content into a list:

```
var task = client.GetAsync("contacts");
HttpResponseMessage response = await task;
response.EnsureSuccessStatusCode();
Stream content = await response.Content.ReadAsStreamAsync();
var serializer = new DataContractJsonSerializer(typeof(List<Contact>));
var contacts = serializer.ReadObject(content);

this.lstContacts.ItemsSource = contacts;
```

The preceding code creates a list of contacts that have been sent from the service. To do that, we got the `Stream` response from the `response` object and we deserialize using `DataContractJsonSerializer`. To use the same thing for a service, which receives XML rather than JSON, you can use `DataContractSerializer` to deserialize.

5. To implement the **Send** button, we use `PostAsync` of the `HttpClient` type:

```
var contact = grdNewContact.DataContext as Contact;
if (contact != null)
{
    var serializer = new DataContractJsonSerializer(typeof(Contact));
    using (var ms = new MemoryStream())
    {
        serializer.WriteObject(ms, contact);
        ms.Position = 0;
        var content = new StreamContent(ms);
        content.Headers.ContentType = new System.Net.Http.Headers.MediaTypeHeaderValue("application/JSON");
        HttpResponseMessage response = null;
        response = await client.PostAsync("contacts", content);
    }
}
```



```
        response.EnsureSuccessStatusCode();
        tbResult.Text = response.StatusCode.ToString();
    }
}
```

In the preceding code, we first created an object of `MemoryStream` to write the object into `Serializer`. The serialized object is then passed to the `PostAsync` method. The serialized object is actually a `StreamContent` which is sent through the network. You can see we need to specify the `ContentType` beforehand to make the service proper idea about the content.

6. Similarly, the delete operation is done using the `DeleteAsync` method of `HttpClient`:

```
var contact = this.lstContacts.SelectedItem as Contact;
if (contact != null)
{
    HttpResponseMessage response = await client.
DeleteAsync("contact/" + contact.Roll.ToString());
    response.EnsureSuccessStatusCode();

    tbResult.Text = response.StatusCode.ToString();
}
```

Here the object that has been selected on the `ListBox` control and is deleted from the server. The `DeleteAsync` calls the HTTP service with the `Delete` verb and the service on the other hand gets the roll of the object and removes it from the list.

7. Tiles are an important section of any application. The default tiles are configured by the application from the `package.appxmanifest` file. On the application UI settings, there is a **Tile** section that allows you to specify various tile settings as shown in the following screenshot:

Tile:	
Logo:	Assets\Logo.png ✕ Required size: 150 x 150 pixels
Wide logo:	Assets\WideLogo.png ✕ Required size: 310 x 150 pixels
Small logo:	Assets\SmallLogo.png ✕ Required size: 30 x 30 pixels
Short name:	Tiles Demo Application
Show name:	Wide Logo Only ▼
Foreground text:	Light ▼
Background color:	#FFFFFF

In the preceding settings pane, you can select a **Logo**, **Wide logo**, and **Small logo** values. The logo will appear when the tile is smaller. The wide logo will be displayed when the tile is set to show a larger view and the small logo appears on the search pane. The required sizes are important to note.

We can also select the background color of the logo, and a short name for the tile which will appear above the time as floating text. The configuration also allows whether the name should be displayed.

8. Tiles can also be made interactive using live updates. There are a large number of XML templates that allow showing a specific order of a live tile. Each live tile has an XML document associated with it.

You can find the entire list of templates at <http://bit.ly/Win8Templ>.

9. Let us add some code inside the `btnGetContact` button such that when the list is updated, we update the tile associated with the application:

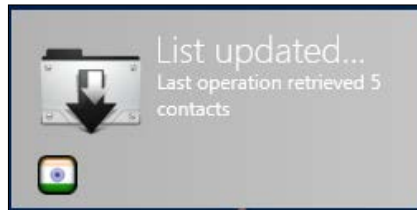
```
private void UpdateRetrieveTile(List<Contact> contacts)
{
    var templateType = TileTemplateType.
TileWideSmallImageAndText04;
    var xml = TileUpdateManager.GetTemplateContent(templateType);
    var textNodes = xml.GetElementsByTagName("text");
    textNodes[0].AppendChild(xml.CreateTextNode("List updated from
server"));
    textNodes[1].AppendChild(xml.CreateTextNode(string.
Format("Last operation retrieved {0} contacts", contacts.Count)));

    var imageNodes = xml.GetElementsByTagName("image");
    var elt = (XmlElement)imageNodes[0];
    elt.SetAttribute("src", "Assets/UpdateList.png");

    var tileUpdater = TileUpdateManager.
CreateTileUpdaterForApplication();
    var tile = new TileNotification(xml);
    tileUpdater.Update(tile);
}
```

In the preceding code, we simply got the `TileTemplate` document from `TileTemplateManager.GetTemplateContent`, which takes an argument of the type of the template that we need to use. After that, we did some simple modifications to `textnodes` and the images to ensure my tile shows the content.

10. If we place the call to the `UpdateRetrieveTile` method after the contact has been retrieved, we will see the tile of our app has been updated on the Start menu:

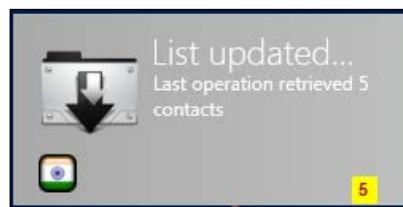


This tile shows an image on the left and two lines of text on the right.

11. Similar to the Tile updates, there is also a special type of badge update that can be used to show notifications to the user. The badge section belongs to the very right-hand corner of a tile:

```
BadgeTemplateType type = BadgeTemplateType.BadgeNumber;
var bxml = BadgeUpdateManager.GetTemplateContent(type);
var element = (XmlElement)bxml.SelectSingleNode("badge");
element.SetAttribute("value", contacts.Count.ToString());
var badge = new BadgeNotification(xml);
var updater = BadgeUpdateManager.
CreateBadgeUpdaterForApplication();
updater.Update(badge);
```

The preceding code updates the badge to the tile, which shows the number of records that have been currently retrieved in a small section to the bottom-right of the tile:



This tile shows the badge in yellow on the bottom-right side of the tile.

12. The toast notification is a flying notification message box that is shown on the screen of the user even though the application is not running. To open a simple toast notification, let us add the following code:

```
var template = ToastTemplateType.ToastImageAndText01;
var xml = ToastNotificationManager.GetTemplateContent(template);
var textNode = xml.GetElementsByTagName("text").FirstOrDefault();
```

```

textNode.AppendChild(xml.CreateTextNode(string.Format("{0}
contacts were retrieved!", contacts.Count)));
var imageNode = (XmlElement)xml.GetElementsByTagName("image").
FirstOrDefault();
imageNode.SetAttribute("src", "Assets/UpdateList.png");
var notification = new ToastNotification(xml);
ToastNotificationManager.CreateToastNotifier().Show(notification);

```

The preceding code creates a `ToastNotification` instance to the user using the predefined template specified in `ToastTemplateType`. The `ToastNotificationManager.CreateToastNotifier().Show()` method creates the `toastNotification` instance and shows it directly on the user screen.

13. When it comes to a toast notification, you can also schedule a toast, such that even when the application stops execution, the toast notification will still appear:

```

var notification = new ScheduledToastNotification(xml, DateTime.
Now.AddSeconds(30));
ToastNotificationManager.CreateToastNotifier().
AddToSchedule(notification);

```

In this case, the toast notification will be displayed 30 seconds after the notification has been invoked.

How it works...

Network access is one of the primary needs for any application. As we are moving more and more towards the Internet era, applications need to update themselves directly from the Internet to give a greater edge to customers. The `HttpClient` class is built on top of the WinRT network API that provides a number of new async methods, which is capable of employing itself in the new async/await pattern. `HttpClient` is capable of producing network calls using normal HTTP protocols for basic HTTP services or RESTful services. In the recipe, we have covered how to invoke requests using all HTTP verbs.

`HttpClient` can also be configured to provide custom header information for the request, shown as follows:

```

HttpRequestMessage request = new HttpRequestMessage();
request.Method = new HttpMethod("POST");
request.Headers.Accept.Add(new System.Net.Http.Headers.MediaTypeWithQu
alityHeaderValue("application/xml"));
request.RequestUri = new Uri("http://localhost:8080/contactservice");

await client.SendAsync(request);

```



Once your service is ready, you need to deploy it to a server and replace the URL of `localhost` here. The service needs to be hosted on a public server before the application can pass certification.

The preceding code configures the bare `HttpRequestMessage` body with accept headers and/or other header information, the method that is to be used to send the request, the request URI, and so on. If we need to handle any specific request structure for our service, the `HttpClient` class is capable of giving us this flexibility as well. The `SendAsync` method actually invokes a network request to the HTTP request URI and gets the response back to us when available. We can also use `WebClient` when we are required to handle web request/response rather than HTTP.

Networking is related to notifications. Windows 8 style tile applications provide three types of notifications that can run out of the box. They are external services that can be invoked even though the application is not running. They are as follows:

- ▶ **Toast:** They are notifications that run for 7 seconds and come over any running application
- ▶ **Tile:** They display information on a tile on the start screen
- ▶ **Badge:** They produce small images over the bottom-right side of a tile

Depending on the importance of the notification, we can use any one of the notifications that a Windows application supports. To invoke a notification, we need to use the manager class associated with the notification type and call its update. They follow XML's standard schema, such that based on the type of notification one selects, it will specify the data for the XML and send the XML directly to invoke it in the environment.

There's more...

HTTP is one of the most important basic needs for any application. The Notification API can be used to create smart applications. Let us discuss some advanced scenarios here.

How to authenticate a web service using Web Authentication Broker API

Authentication is one of the important concerns for any website. Many of the apps we create for Windows 8 style applications are generally to communicate to external services. As with growing OAuth techniques, most external web applications support the open authentication protocol. Windows 8 style application exposes a `WebAuthenticationBroker` class, which implements the OAuth itself such that you do not need to manually create the Open Auth protocol.

Facebook gives OAuth support that we can use to demonstrate the authentication. Let us create an app in the `www.developers.facebook.com` page and get an `appId` value for your application. Once you have that you can implement the OAuth for your application and get the friend list of all your friends:

```
HttpClient client = null;
private string appId = "000000000000000";
private string loginUri = "https://www.facebook.com/dialog/oauth";
private string redirectUri = "https://www.facebook.com/connect/login_
success.html";

private string authToken = "";
bool isAuthenticated = false;

public FacebookClient()
{
    client = new HttpClient();
    client.MaxResponseContentBufferSize = 100000;
    client.BaseAddress = new Uri("https://graph.facebook.com");
}

private async Task AuthenticateAsync()
{
    var requistr = string.Format("{0}?client_id={1}&redirect_
uri={2}&response_type=token",
        loginUri, appId, redirectUri);
    var requestUri = new Uri(requistr, UriKind.RelativeOrAbsolute);
    var redirectionUri = new Uri(redirectUri, UriKind.
RelativeOrAbsolute);

    var result = await WebAuthenticationBroker.AuthenticateAsync(WebAu
thenticationOptions.None, requestUri, redirectionUri);
    if(result.ResponseStatus != WebAuthenticationStatus.Success)
        throw new Exception("Login failure : " + result.
ResponseErrorDetail);

    authToken = GetAuthTokenFromResponse(result.ResponseData);
    if (!string.IsNullOrEmpty(authToken))
        client.DefaultRequestHeaders.Authorization = new System.Net.
Http.Headers.AuthenticationHeaderValue("OAuth", authToken);
    isAuthenticated = true;
}
```

```
private string GetAuthTokenFromResponse(string str)
{
    return str.Split(new string[] { "access_token=", "&expires_in" },
        StringSplitOptions.None) [1];
}
```

In the preceding code, we have created a request on the URL. Facebook provides a request URL for login, and a redirect URL that the page will be redirected to once the login is successful. In the code we have prepared the authentication URL for the Facebook page and passed it to `WebAuthenticationBroker.AuthenticateAsync`. This method takes three arguments, the first being the options for the protocol, the second is the authentication URI, and the third is the redirect URI. The call will automatically open a pop up on the screen and create an authentication for Facebook. After you log in to the Facebook page, you will be redirected to your app.

We are using the graph API to get the list of friends from the Facebook account. To access a friend list using the graph URL, we need the authentication token. Once the authentication is made, we receive the token result as a string in `result.ResponseData`. We have parsed and extracted the token for future use:

```
ObservableCollection<KeyValuePair<string, string>> _friends;
public ObservableCollection<KeyValuePair<string, string>> Friends
{
    get
    {
        _friends = _friends ?? new ObservableCollection<KeyValuePair<string, string>>();
        return _friends;
    }
}
public async Task GetFriends()
{
    try
    {
        if (!this.isAuthenticated)
            await this.AuthenticateAsync();

        var stream = await client.GetStreamAsync("me/
friends&fields=first_name,last_name");
        var serializer = new DataContractJsonSerializer(typeof(List<KeyValuePair<string, string>>));
        var friends = serializer.ReadObject(stream) as
List<KeyValuePair<string, string>>;
    }
}
```

```

        Friends.Clear();
        foreach (var f in friends)
            Friends.Add(f);
    }
    catch { }
}

```

To access the friend list, we have invoked the `GetStreamAsync` method on the Facebook graph API. If the authentication is successful, the graph API will get the list of all friends that are associated with the current user.

The `WebAuthenticationBroker.AuthenticateAsync` method is an unified API that has been provided to any Windows 8 style application for open authentication.

Even though I have been using `WebAuthenticationBroker` to connect to Facebook, there is also a well-defined C# SDK for Facebook that works on the same logic from inside but has a better API and is easier to use. Please use it when you want a target application for Facebook.

Refer to the link <http://bit.ly/FacebookSDK>.

How to implement Push notifications in a connected Windows 8 style tile application

Windows notification service is a highly-scalable notification service that sends notifications to billions of users at a time. The WNS is a live cloud service that allows you to register a channel directly to it and it notifies the app when there is any update available for it. The WNS automatically invokes the Push notification message to the clients, such that only when the data is available or only when the server wants it, it can initiate the connection to the clients. Before discussing the Push notification, let us consider the following code:

```

public void PeriodicTrigger()
{
    var tileupdater = TileUpdateManager.
        CreateTileUpdaterForApplication();
    tileupdater.StartPeriodicUpdate(new Uri(string.Format("{0}{1}",
        uri, "tile")), PeriodicUpdateRecurrence.HalfHour);
}

```

In the preceding code, a periodic notification has been created and the registered app will call the URI at an interval of `HalfHour` and update the tile. Now say for instance, you have thousands of apps running. This periodic call to the server may kill your server's bandwidth. Also if it has some update in the first minute on the server, it has to wait for 29 minutes more to get the same update. Hence, it would be nice if only when the data is available can the server initiate the notification itself.

To implement the Push notification we need to first register the app to WNS service. Once the registration is done, the server can push messages directly through WNS. Let us add a method to register the service. On the server side, we create a service and specify one service endpoint to register the app. The code looks like the following:

```
public void RegisterPushService(PushData data)
{
    if (data == null) return;
    Uri uri = null;
    if (!Uri.TryCreate(data.ChannelUri, UriKind.Absolute, out uri))
    {
        throw new WebFaultException<string>("Invalid uri", System.Net.
        HttpStatusCode.BadRequest);
    }
    var dns = uri.Authority;
    if (!dns.EndsWith("notify.windows.com"))
        throw new WebFaultException<string>("Invalid domain", System.
        Net.HttpStatusCode.BadRequest);

    registeredClients.Add(data.UniqueId, uri);
}
```

The code is very simple. We first validate the URI to check whether it is a valid WNS URI. The URI that supports WNS ends with `notify.windows.com`, hence we can check the URI to send fault exceptions from the service. Once the app is validated, it creates a dictionary of ID and URI. The ID is a GUID specified on the client to uniquely identify the app.

On the client side, we need to register the app when it is launched. Let us consider the following code:

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    try
    {
        channel = await PushNotificationChannelManager.
        CreatePushNotificationChannelForApplicationAsync();
    }
    catch { }
    var data = new PushData
    {
        ChannelUri = channel.Uri,
        UniqueId = App.ClientID
    };
    await SendDataToServiceAsync<PushData>(data, "register");
}
private async Task SendDataToServiceAsync<T1>(PushData data, string
uri)
{

```

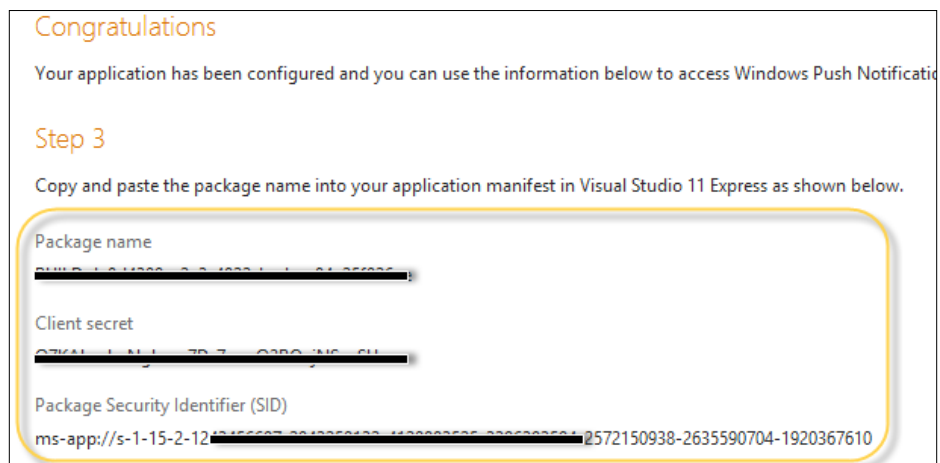
```

StringContent contnet = null;
using (var ms = new MemoryStream())
{
    var ser = new DataContractSerializer(typeof(T1));
    ser.WriteObject(ms, data);
    ms.Position = 0;
    contnet = new StringContent(new StreamReader(ms).ReadToEnd());
}
contnet.Headers.ContentType = new System.Net.Http.Headers.
MediaTypeHeaderValue("text/xml");
var response = await client.PostAsync(uri, contnet);
response.EnsureSuccessStatusCode();
}

```

On the `NavigatedTo` method, we process `App` to create a WNS channel. The `PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsync` method creates a new channel with the WNS URI. We can use the same to register our app to WNS. We create a `HttpClient` call to our RESTful service created on the server to call the `RegisterApp` method and send the `channeluri` value and the client's unique identification to register the client.

Now to use WNS from our own server (or our own WCF service) we need to first get credentials from the Microsoft Windows Dev Center Dashboard, which we can only get when we submit our app to the Windows Store and it is basically associated with the production process. But Microsoft also provides a test management portal to generate the same thing without submitting the app to try out for test. Let's open the URL: <https://manage.dev.live.com/build>. After logging into the portal using your live ID, you will see a page which specifies how to use the service. Open your package.appxmanifest file and go to the **Packaging** tab and copy the package display name and the publisher name to the online portal and accept. It will provide the package name, the secret keys, and identifier for you:



The **Package name** value needs to be replaced to with the original package name specified in the `Package.appxmanifest` file and we will use the **Client secret** value and the **Package Security Identifier** number later from our app.

The WNS uses OAuth 2.0 and hence we need to use the secret credentials to authenticate with the service. Let us use WNS live service to push some data to our app and notify using tile or toast notification.

Let us first register our app with the service:

```
string sid = "Q7KakzubaNgbme7D-7mwQ2BOvjNSsoSU";
string secret = "ms-app:/
/s-1-15-2-1243456607-2043250132-4120883525-3396393594-2572150938-
2635590704-1920367610";
string authuri = "https://login.live.com/accesstoken.srf";
string accesstoken = string.Empty;
private void RegisterWNS()
{
    var body = string.Format("grant_type=client_credentials&client_
id={0}&client_secret={1}&scope=notify.windows.com",
        HttpUtility.UrlEncode(sid), HttpUtility.UrlEncode(secret));
    var client = new WebClient();
    client.Headers.Add("content-type", "application/x-www-form-
urlencoded");

    string response = client.UploadString(new Uri(authuri), body);
    var ser = new DataContractJsonSerializer(typeof(AccessTokenRespon
se));
    using (var stream = new MemoryStream(Encoding.UTF8.
GetBytes(response)))
    {
        AccessTokenResponse tokenResponse = (AccessTokenResponse)ser.
ReadObject(stream);
        accesstoken = tokenResponse.AccessToken;
    }
}
```

In the preceding code we have created a web request to point to `authuri` and get the `accesstoken` value that is received as response. As WNS uses OAuth, we need to specify the body of the request with a special value as defined at `http://bit.ly/PushResponseHeaders`. It receives a JSON response and we parse it to get the `accesstoken` value. We have also stored the access token to a safe variable such that we can re-use the same later on.

Once we have the header, we can go on and send the Push notification as shown in the following code:

```
private void NotifyRegisteredClients(string message)
{
    var templateName = "ToastImageAndText01";
    var toast = string.Format("<toast>" +
                             "<visual>" +
                             "<binding template=\"ToastImageAn
dText01\">" +
                             "<image id=\"1\"
src=\"Assets\\metroNotification.png\" alt=\"Special Notification\"/>"
+
                             "<text id=\"1\">{0}</text>" +
                             "</binding>" +
                             "</visual>" +
                             "</toast>", message);
    foreach (var uri in this.registeredClients.Values)
    {
        SendNotificationToClient(uri, toast);
    }
}

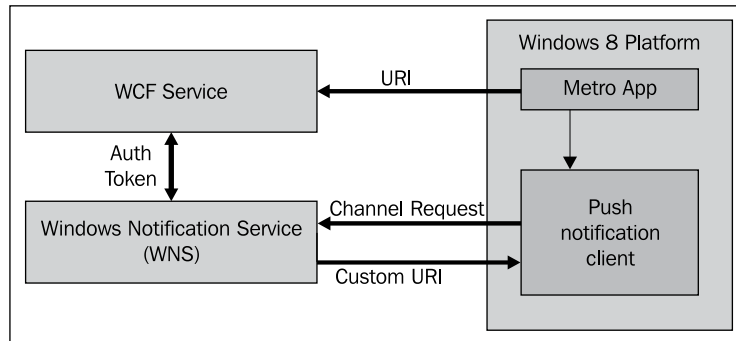
private string SendNotificationToClient(Uri uri, string toast)
{
    byte[] content = Encoding.UTF8.GetBytes(toast);
    string status = "";

    var request = HttpWebRequest.Create(uri) as HttpWebRequest;
    request.Method = "POST";
    request.Headers.Add("X-WNS-Type", "wns/toast");
    request.Headers.Add("Authorization", string.Format("Bearer {0}",
accesstoken));

    request.BeginGetRequestStream(result =>
    {
        var requestStream = request.EndGetRequestStream(result);
        requestStream.Write(content, 0, content.Length);
        request.BeginGetResponse(e =>
        {
            var response = request.EndGetResponse(e) as
HttpWebResponse;
            status = response.StatusCode.ToString();
        }, null);
    }, null);
    return status;
}
```

In this code I have created a toast notification and sent the toast through WNS. The message passed, will be sent to the actual app that has been registered with the WNS. The WNS broadcasts our message to the app through the secured channel that has been created.

Finally, if we invoke the `NotifyApp` URL from any application, this will notify the message to all the apps that have registered with our service. You can send tile or badge notifications in the same way as we did for toast notification.



The preceding diagram shows the entire architecture of the Push notification service. The Windows 8 platform creates an instance of the Push notification client, which is used to register itself with WNS. The Push notification client sends a channel request and gets a custom URI as response. This URI is used to create a secure channel between the app and the WNS. Now the app registers the custom WCF service using the client URI, such that the WCF service in turn gets an authorization token with the WNS. Now when something needs to be notified to the client, the WCF service can invoke that request to WNS using the channel token registered by the Push notification client in the app.

So basically, the WCF gets an authorization token and sends a request to an already registered URI of the client and the whole process gets executed. Once the notification is received by the client from the WNS system, the Windows 8 environment can invoke toast, tile, or badge notifications.

How to use Syndication API and AtomPub API for handling feeds in Windows 8 style application

If you are working with feeds, the WinRT API exposes a class that can help you retrieve the feeds. The Syndication API is constructed to retrieve a strongly-typed reference object from the XML Atom- or RSS-based feeds. So as a developer, you can make use of the classes to get the Atom and RSS feeds directly without manually parsing the XML document.

`SyndicationClient` is a `HttpClient` object that helps in getting Syndication feeds from the Atom or RSS feed passed to it. Let us look into some code:

```
var client = new SyndicationClient();
SyndicationFeed feed = await client.RetrieveFeedAsync(new
Uri(atomurl));

feedData.Title = feed.Title.Text;
foreach (SyndicationItem item in feed.Items)
{
    FeedItem feedItem = GetFeedItem(item, feed.SourceFormat);
    feedData.Items.Add(feedItem);
}
```

In the preceding code, the `SyndicationClient.RetrieveFeedAsync` method is called, which will get a strongly-typed object of `SyndicationFeed` that has all information about the feed it receives. The `Title` attribute specifies the title of the blog and `Feed.SourceFormat` specifies the format of the feed, and so on:

```
private FeedItem GetFeedItem(SyndicationItem item, SyndicationFormat
syndicationFormat)
{
    var feedItem = new FeedItem();
    feedItem.Format = syndicationFormat;
    feedItem.Title = item.Title.Text;
    feedItem.PubDate = item.PublishedDate.DateTime;
    if (item.Authors.Count > 0)
        feedItem.Author = item.Authors[0].Name;
    if (syndicationFormat == SyndicationFormat.Atom10)
        feedItem.Content = item.Content.Text;
    if (syndicationFormat == SyndicationFormat.Rss20)
        feedItem.Content = item.Summary.Text;

    if (item.Links.Count > 0)
        feedItem.Link = item.Links.FirstOrDefault().Uri;

    return feedItem;
}
```

The data can be retrieved from a strongly-typed object. For instance, the title of a blog can be retrieved from its `Title` attribute, then we can use `PublisherDate`, `Authors` (if any), and so on.

The AtomPub API

In addition to retrieving the feeds from the online web content, Windows 8 also provides inbuilt support of the AtomPub API that helps in updating the Atom-based feeds. You can post new content to the Atom feed URL by providing the credentials that are required to log in. Once the service is authenticated properly, you can use the AtomPub API directly to post your blog or article to the online web content:

```
public async void CreatePost(string title, string summary, string
content)
{
    var cred = new PasswordCredential();
    cred.UserName = "abhi";
    cred.Password = "";

    var atomClient = new AtomPubClient(cred);
    string fullposturl = "http://abhisheksur.wordpress.com/wp-app.php/
posts"
    var post = new SyndicationItem();
    post.Title = new SyndicationText(title);
    post.Summary = new SyndicationText(summary);
    post.Content = new SyndicationContent(content,
SyndicationTextType.Text);

    var author = new SyndicationPerson("Abhishek");
    post.Authors.Add(author);

    await atomClient.CreateResourceAsync(new Uri(fullposturl), title,
post);
}
```

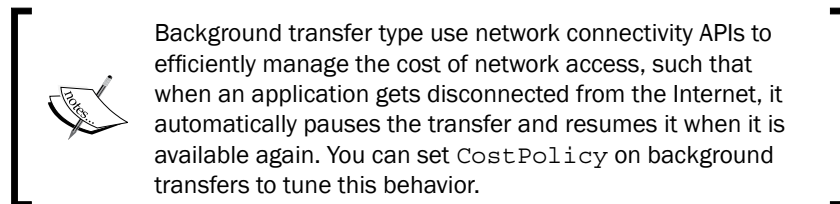
The preceding code uses `AtomPubClient` to post a blog on the online blog created. The `PasswordCredentials` class takes the username and password that need to be passed to the `AtomPubClient` class to ensure that the request is well authenticated. It adds them to every request header. We create an object of `SyndicationItem` and pass the title, summary and content of the blog, and add authors to the blog and finally use `CreateResourceAsync` to post the object to the link.

How to perform background transfers of data in Windows 8 style tiles applications

For long running downloads or uploads, `HttpClient` gets requests that are not sufficient enough to handle. Windows 8 style apps are subjected to its life cycle events, and thus when the transfers are long running, the application life cycle can pretty much hamper the transfer operation.

For instance, while downloading data from server, if the user switches between apps, the app that launches the transfer may go to a dormant stage called **suspended**, where all threads on the app will get blocked and hence any transfer that has been invoked by the application will also stop. Thus big file downloads cannot be handled by the foreground transfer techniques.

To overcome the situation, Windows 8 apps provide a separate process called **background services** that can be invoked from any app to handle data transfers in the apps, such that the foreground runs independently from the background transfer and any life cycle changes to the app will not affect the background transfer. Background transfers support HTTP, HTTPS, and FTP transfer and support the pause/resume ability.

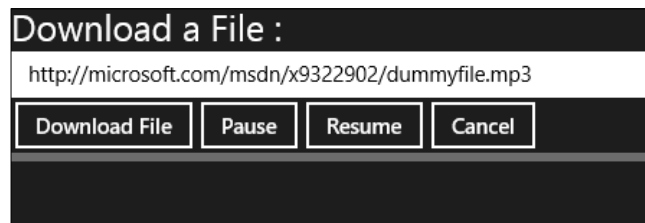


Background transfers are performed using the `BackgroundUploader` or `BackgroundDownloader` class inside the WinRT API. Both the classes implement the `IBackgroundTransferBase` interface of WinRT and is defined in the `BackgroundTransfer` namespace.

How to do it...

The following steps take you through background transfers:

1. Let us define a UI first. We create page (let us use the default page that is created with the project template) and call it `MainPage.xaml`. We place one `TextBox` control called `txtFile` to take the input from the user. We also create four buttons to download a file, pause, resume, and cancel respectively. We also place a `ProgressBar` control named `prgFileProgress` to show the actual progress of the download operation. The following screenshot illustrates the layout:



In the preceding application, we use the button **Download File** to initiate the downloading operation of the file specified on the `txtFile` textbox. The progress is shown in the `Progressbar` control from standard progress callback notification.

2. Let us look into the code as to find out how to implement background transfers:

```
SynchronizationContext context;
IAsyncOperationWithProgress<DownloadOperation, DownloadOperation>
asyncOperation;
DownloadOperation downloadOperation;
private async void btnDownload_Click(object sender,
RoutedEventArgs e)
{
    string path = txtfile.Text;
    await StartDownloadAsync(path);
}
internal async Task StartDownloadAsync(string filePath)
{
    var downloader = new BackgroundDownloader();
    var fileName = Path.GetFileName(filePath);
    var storageFile = await KnownFolders.MusicLibrary.
CreateFileAsync("test\\" + fileName, CreationCollisionOption.
ReplaceExisting);

    downloadOperation = downloader.CreateDownload(new
Uri(filePath), storageFile);

    asyncOperation = downloadOperation.StartAsync();
    context = SynchronizationContext.Current;

    asyncOperation.Progress = new AsyncOperationProgressHandler<Do
wnloadOperation, DownloadOperation>(notifyProgress);

    await asyncOperation;
}
```

In the preceding code, when the **Download** button is clicked on by the user, an object of `BackgroundDownloader` is created and `CreateDownload` is called from it. The `CreateDownload` method takes a reference of the URI to which it needs to download and the `storageFile` reference where the downloader will download. We have marked the code required to start the download operation.

3. The `CreateDownload` method also requires a reference to a `storageFile` object reference so that it can download to that location. In the code, I have used the `MusicLibrary` folder of the disk. We take the reference from an already existing API defined as `KnownFolders.MusicLibrary`.



To use any external folder, you also need to add an entry to the manifest file. Here, `MusicLibrary` is used and hence we need to add the `MusicLibrary` capability in the manifest of the project. Also, if we are using `FilePicker`, we need to add a `FilePicker` declaration too.

4. When the `StartAsync` method is called from the `DownloadOperation` object, the actual download gets started. The `StartAsync` method runs on a separate process and we can await it.
5. We store `SynchronizationContext.Current` to make sure we always post on the right thread to show progress. The `BackgroundDownloader` class works on a separate process inside the environment and the callback might come from a separate thread than the UI thread.
6. We also hook the progress download operation by passing the delegate pointing to `notifyProgress`, such that it will be called to update the progress in the UI. To update the UI we use the following code:

```
void notifyProgress (IAsyncOperationWithProgress<DownloadOperation,
DownloadOperation> asyncOp, DownloadOperation downloadOp)
{
    context.Post (state =>
    {
        this.prgFileProgress.Maximum = downloadOp.Progress.
TotalBytesToReceive;
        this.prgFileProgress.Value = downloadOp.Progress.
BytesReceived;
    }, null);
}
```

The callback `notifyProgress` method is passed to the `Progress` delegate of the `IAsyncOperationWithProgress` WinRT interface to hook the method that notifies the progress of the file download. Remember, as the `Progress` delegate is called in on a non-UI thread, we need to hold the `SynchronizationContext` class beforehand to ensure the call to the UI object is made from `Dispatcher`.

7. When the application has already started downloading the files, you can cancel this operation, or even pause/resume the download using the following code:

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
{
    if (asyncOperation != null && asyncOperation.Status ==
AsyncStatus.Started)
    {
        asyncOperation.Cancel();
    }
}
```

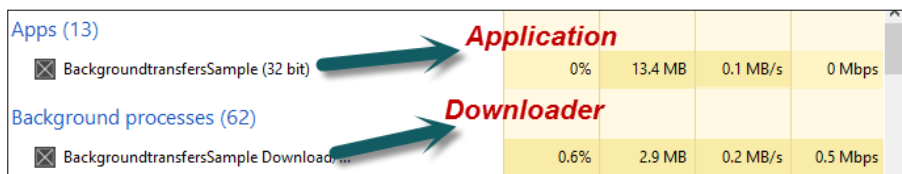
In the preceding code, the `IAsyncOperationwithProgress` interface has been held by the application, and used to cancel a running download.

8. Similarly, the `DownloadOperation` class also exposes methods to pause or resume the transfer, provided we are in the state to do that. Let us look at the following code:

```
private void btnPause_Click(object sender, RoutedEventArgs e)
{
    if(downloadOperation != null && downloadOperation.Progress.
    Status == BackgroundTransferStatus.Running)
    {
        downloadOperation.Pause();
    }
}

private void btnResume_Click(object sender, RoutedEventArgs e)
{
    if (downloadOperation != null && downloadOperation.Progress.
    Status == BackgroundTransferStatus.PausedByApplication)
    {
        downloadOperation.Resume();
    }
}
```

Here the `btnPause_Click` method calls `Pause`, while `btnResume` calls the `Resume` method. It is important to note that we should check the status of the ongoing operation before calling these methods.



	Application			
BackgroundtransfersSample (32 bit)	0%	13.4 MB	0.1 MB/s	0 Mbps
Background processes (62)				
BackgroundtransfersSample Download...	0.6%	2.9 MB	0.2 MB/s	0.5 Mbps

9. When we start `BackgroundTransfer`, the application creates a new process to invoke the transfer. You can open the Task Manager in Windows and see that there are two processes listed (as shown in the preceding screenshot). Even though the main application gets suspended, the background process will still remain active and continue. Now sometimes, we know during the process life cycle, the application may terminate execution. `BackgroundTransfer` also provides an API to reattach existing ongoing transfers when an application is relaunched with an ongoing transfer still remaining:

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    context = SynchronizationContext.Current;
```

```

        await attachPendingDownload();
    }

    private async Task attachPendingDownload()
    {
        List<Task<DownloadOperation>> tasks = new List<Task<DownloadOp
eration>>();
        var downloads = await BackgroundDownloader.
GetCurrentDownloadsAsync();

        if (downloads.Count > 0)
        {
            downloadOperation = downloads[0];
            txtfile.Text = downloadOperation.RequestedUri.ToString();
            asyncOperation = downloadOperation.AttachAsync();
            asyncOperation.Progress = new AsyncOperationProgressHandle
r<DownloadOperation, DownloadOperation>(notifyProgress);

            await asyncOperation;
        }
    }
}

```

In the preceding code, the `BackgroundDownloader`.
`GetCurrentDownloadsAsync` method gets a list of all the background transfer files
that are currently downloading. In our case, I am just holding the first file that is getting
downloaded, and the information is retrieved by the `onNavigate` property of the page.
We have also hooked the `notifyProgress` delegate to the `DownloadOperation`
object so that it notifies the application progress bar with an actual seed value.

10. Moreover, you can define `CostPolicy` for your downloader to ensure that
in a restricted network we do not consume too much bandwidth. It takes a
reference of `BackgroundTransferCostPolicy`, which can be `default`,
`unrestrictedOnly`, or `always`:

```
downloadOp.CostPolicy = BackgroundTransferContPolicy.always;
```

`default` allows transfer in costed networks, while `unrestrictedOnly` will not.

Similar to download, `BackgroundUploader` is used to upload files. We can use
the `CreateUpload` method to initiate the upload operation almost identical to the
download, and call `Push` to push the upload. Unlike `BackgroundDownloader`,
`BackgroundUploader` does not support the pause/resume functionality if the
upload has been disconnected.

How it works...

Background transfer works in a separate process outside the domain of the calling environment. When the app is suspended, the download can still continue in the background. The operating system automatically determines when to stop/pause operations based on the policy setup while the transfer is invoked. To ensure that the application always gets the latest updated status, the `BackgroundDownloader` class has a `GetCurrentDownloadsAsync` method, which invokes the background downloader process and gets the status of the download operation. The Task Manager of Windows also shows the status of these operations in a separate process.

Debugging in Visual Studio works differently. Stopping a debugging session means terminating an application. In this case the downloads are paused. While debugging the application also enumerates pause, resume, or cancel of any persisted download.

See also

- ▶ Refer to the following link:
<http://bit.ly/BackgroundTransfer>

Index

Symbols

96DPI screen 224
120DPI application 224
120DPI settings 224
#BLOB format 58
#GUID format 58
.NET
 about 7
 lazy objects 110
 weak references 108, 109
.NET assembly
 internal structure, inspecting 53-58
 types 60-63
.NET framework
 components 52, 53
.NET program
 components, inspecting 70-72
 memory leaks, searching in 105-108
.pdb file 56
#~Stream format 58
#String format 58
#US format 58

A

Abort method 127
Activate method 329
Adobe AIR (Flash) 190
AJAX 189, 209
AlternateltemTemplate property 186
animation
 about 291
 enabling, within Windows 8 style tiles
 application 291, 292
 implementing, in Windows 8 style tiles
 applications 304, 305

app

 enabling, for app sharing 326-329

AppDomain

 about 131
 code, isolating 115

application

 building, MVVM pattern used 234-245
 debugging 28, 29
 packaging 321, 323

application cache, HTML5

202, 203

application configuration file

74

application domains

131

application life cycle, WinRT applications

312-320

application manifest file

73, 74

Application Model

277

App section, XAML window

302

appxupload file

323

Architecture Explorer tool

24

A Sharp

53

Asmex

 about 58

 URL 58

ASP.NET

 about 173

 working with jQuery, in Visual Studio 205-210

ASP.NET 4.5

 configuration changes 222

ASP.NET applications

 CSS3 190-196

 HTML5 190-196

 statically-typed model binding 185-188

ASP.NET servers, DLL intern

185

ASP.NET web applications

 performance boosters 175-181

ASP.NET websites

compilation of pages, configuring 182

assemblies

merging 88

assemblies, WPF framework

PresentationCore.dll 227

PresentationFramework.dll 226

WindowsBase.dll 227

assembly

about 53

configuring, from Visual Studio 94, 95

delay signing 66, 67

disassembling 81-86

disassembling, Reflector used 87, 88

inspecting, assembly viewer used 58, 59

obfuscating, steps 90-92

assembly file

sections 57

AssemblyInfo file 72

assembly linker 53

AssemblyMetadataAttribute 73

assembly viewer

used, for inspecting assembly 58, 59

async

about 211

anonymous method, writing 164

working with 155-164

AsyncCompletedEventArgs parameter 136

AsyncDelegate property 129

asynchronous programming 124

Asynchronous Programming Model (APM) 127

asynchronous programming, patterns

Asynchronous Programming Model (APM) 127

Event-based Asynchronous Pattern 127

Task-based Asynchronous Pattern (TPL) 127

asynchronous threading pattern 126-130

async language 134

AsyncResult, properties

AsyncState 130

AsyncWaitHandle 130

CompletedSynchronously 130

IsCompleted 130

AsyncState property 130

AsyncWaitHandle property 130

AtomPub API

about 348

used, for handling feeds in Windows 8 style

application 346, 347

audio tag 195

authentication 338

AutoResetEvent 131

awaitable methods 165

await patterns

working with 155-164

B

background GC 99

background services 349

background transfers of data

performing, in Windows 8 style tiles

applications 348-354

BackgroundWorker

about 135

working with 136-139

BackgroundWorker class 139

BAML 232

Barrier class

working 132, 133

Begin pattern 124

Binary XAML. *See* BAML

binding 185, 251

Binding Expression

about 251

uses 251-254

Blank App 279

blocking methods 140

blocks

linking 170, 171

Boo 53

Bookmark menu 26

BSJB 56

Buffering Blocks 166

Bundle.GetBundleResponse method 181

Bundle.ProcessRequest method 181

BundleResponse class 180

bundling

benefits 177-180

Button control 332

C

C#

- about 7, 155
- used, for building Windows 8 style tiles application 297-302

C++ 7

CallNormalAsyncMethod 128

CameraCaptureUI class 312

CanExecuteChanged event 237

CanExecute method 237

CheckBox 225

CIL 52

Class View

- working with 15, 16

Clone Code Detection 49, 50

CLR 7, 52

CLS 52

code

- isolating, AppDomain used 115
- securing from reverse engineering, with obfuscation 89-93

Code Definition window 26, 27

code developer

- common mistakes 115-121

Code Highlighting feature

- about 22
- using, in Visual Studio 22, 23

code snippets

- about 39
- using, in Visual Studio 39-42

CoerceValue 249

CollectionView

- Current Record Manipulation 259
- filtering 259
- grouping 258
- Live Shaping 259
- sorting, applying 258
- working with 257

CommandsRequested event 311

command switches, Visual Studio 13, 14

common intermediate language. *See* CIL

Common Language Runtime. *See* CLR

Common Language Specification. *See* CLS

compilation of pages

- configuring, in ASP.NET websites 182

CompletedSynchronously property 130

components

- inspecting, of .NET program 70-72
- of Visual Studio IDE 8-13

concurrent programming 147

configuration

- about 75
- modifying, at runtime 79

configuration changes, ASP.NET 4.5 222

ConfigurationManager API 79

configuration options, TDF blocks

- about 169
- BoundedCapacity 170
- CancellationToken 170
- Greedy 170
- MaxDegreeOfParallelism 170
- MaxMessagePerTask 170
- TaskScheduler 170

configuration versions

- dealing with 79-81

ContentPresenter 225

COR20 header 57

CountdownEvent 131

CSS

- about 189
- used, for building Windows 8 style tiles application 278-290

CSS3 190-196, 223

CSS Editor

- updates 218-221

CTS 53

custom configuration

- working with 75-78

CustomParameters tag 37

D

DataBinding 251

DataBind method 185

data persistence 306

DataRequested event 327

DataSource property 185, 186

DataTransferManager.DataRequested event 326

data validation blocks

- implementing, in MVVM 254-257

data-win-control attribute 285

DebugDiag 104

Debug directories 56

delay signing 66

DeleteAsync method 334

DependencyObject 225 234

DependencyObserver 225

Dependency property

about 245

advantages 246

declaring 245

using 246-249

Device Drivers 227

Device Independent Pixel (DPI) 223, 224

Direct3D 227

DirectX 223

disassembling 83

DLL intern

in ASP.NET servers 185

DocType element 191

Dotfuscator

about 89, 92

options 94

Dotfuscator, options

control flow obfuscation 94

instrumentation 94

Obfuscated code, debugging 94

pruning 94

renaming 94

string encryption 94

Dots per inch (DPI) 223

dynamic link libraries 54

E

EEClass 99

Emit Debug Symbols property 95

End pattern 124

enhancements, Visual Studio editors 214-221

enhancements, WPF

in .NET4.5 226-232

EnsureSuccessStatusCode method 333

entry point 72

enumeration options, Dependency property

AffectsArrange 249

AffectsMeasure 249

AffectsParentArrange 249

AffectsParentMeasure 249

AffectsRender 249

BindsTwoWayByDefault 249

Inherits 249

NotDataBindable 249

Event-based Asynchronous Pattern 127

about 135

working with 136-139

event life cycle

working, in WinJS 292, 293

evolution, Visual Studio 8

Execute method 237

execution engine suspension 98

Executor Blocks 166

Expression Blend tool 225

Extension Manager 27

F

Facebook 326, 339

Fantom 53

features, jQuery 205, 206

feeds

handling, AtomPub API used 346, 347

handling, Syndication API used 346, 347

Figure tag 192

files

previewing, in Visual Studio IDE 21

filter operations, ModelBinding 188, 189

Finalize method 103

finalizer 103

finalizer thread 96

Fixed Layout App 279

footer tag 192

form control 332

Frame section, XAML window 302

friend assembly

creating 64, 65

FromCurrentSynchronizationContext method 153

G

garbage collection

about 96, 98

working 99-102

garbage collection, terms

- managed heap 97
- stack 97
- unmanaged heap 97

GC collection 98

GDI 227

General HTTP API 330

Generate Sequence Diagram option 25

GeoLocation

- getting, HTML5 used 199

GetAsync method 333

GetCurrentDownloadsAsync method 354

GetStreamAsync method 341

GetType method 102

Global.asax file 179, 183

Global Assembly Cache (GAC)

- about 60
- using 69

Grid App 279

GridView 215

H

header tag 192

home.aspx file 178

hostDiv tag 282

HTML5

- about 190-196, 223
- application cache, using 202, 203
- local IndexDB storage 199-202
- notifications, on browsers 204, 205
- sockets 197-199
- used, for building Windows 8 style tiles application 278-290
- used, for detecting online status of browser 203, 204
- used, for getting GeoLocation 199
- web workers 196, 197

HTML editor

- updates 215-217

HTTP 338

HttpClient class 333, 337

HttpHandlers class 211

HttpModules class 211

Hub pages 279

I

ICommand interface

- about 236
- CanExecuteChanged event 237
- CanExecute method 237
- Execute method 237

IDE editors 17

IDE search box 20, 21

IDE workspace

- windows, docking 20

IDispatch interface 276

iFrame 189

IEnumerable<T> interface 276

IL Disassembler 81

ILMerge 88

IL Weaving 89

images

- previewing, in Solution Explorer 16

IndexDB 199

INotifyPropertyChanged event 236

input tag 193

Instance Count

- Mutex, using for 144

instrumentation 94

Integrated Development Environment (IDE) 8

IntelliSense 39

internal infrastructure

- inspecting, of .NET assembly 53-58

InternalsVisibleToAttribute attribute 64

InvokeAsync method 229

IsAlive property 126

IsCompleted property 129, 130

ISourceBlock 170

IsPreviewMouseDown event 234

ItemTemplate property 186

item templates 29

ItemType property 186

J

J# 7

JavaScript

- used, for building Windows 8 style tiles application 278-290

JavaScript editor
updates 217, 218

JIT 104

JIT compiler 53

Joining Blocks 166

Join method 127

jQuery

about 205

features 205, 206

library, extending 210

working 210

just-in-time. *See* **JIT compiler**

L

lambda expression

writing 164

large object heap (LOH) 98

layouts, Windows 8 style tiles applications

Filled 302

FullScreen 302

Landscape 302

Portrait 302

Snapped 302

lazy class 111, 112

lazy Initialization

lazy class 111, 112

LazyInitializer 111-114

ThreadLocal 111-113

LazyInitializer 111-114

lazy objects 110

library

writing, for WinJS 294-296

LinkTo method 170

ListBox control 332

ListBoxItem property 231

ListBox property 231

Live Shaping

used, for working with CollectionView 257

loaded() function 281

local IndexDB storage, HTML5 199, 201, 202

locking

about 139

with Spinlock 145, 146

locking constructs 140

lock statement

task-based application, using in concurrent

programming 148-152

Logical Trees 233

Lsharp 53

M

managed code 52

managed heap 97

managed layer, WPF 226

Managed Profile Guided Optimization. *See*
MPGO

manifest 56

ManualResetEvent 131

Markup Extensions 229

Media Integration Library Core. *See* **milcore**

memory dump files

creating, Visual Studio used 114

memory leaks

about 104

searching, in .NET program 105-108

memory management 96

MessageBox

CoerceValue 249

PropertyChanged 249

ValidateCallback 249

MethodDesc 99

methods, SynchronizationContext class

Post 134

Send 134

MethodTable 99

Microsoft 7, 275

Microsoft Intermediate Language. *See* **MSIL**

Microsoft Windows Dev Center Dashboard

343

milcore 227

MODEL 234

ModelBinding

with filter operations 188, 189

Model-View-ViewModel. *See* **MVVM**

Modernizr scripts 191

MouseDown event 234

MPGO

about 103

used, for optimizing native images 103, 104

MSBuild

- about 27
- using 28

MS-DOS Header information 57

MSIL 7

Multicore JIT 104

Mutex

- about 143
- determining, of thread 143, 144
- using, for Instance Count 144

MVVM

- about 234
- data validation blocks, implementing 254-257

MVVM pattern

- about 234
- used, for building applications 234-245

MZ header information 57

N

Native Image Generation tool. *See* NGen

native images

- optimizing, MPGO used 103, 104

NavigatedTo method 343

navigate method 282, 298

Navigation App 279

nav tag 192

networking 325

network latency 175

NGen 103

non-blocking synchronization constructs 140

non-blocking synchronization constructs, ReaderWriterLock 146, 147

notifications

- working with 330-338

notifications, HTML5 204, 205

Nuget package manager 177

O

OAuth 338

obfuscation

- about 89
- used, for securing code 89-93

OnCheckPoint event 293

online status, of browser

- detecting, HTML5 used 203, 204

OnLoaded method 293

OnSearchActivated method 329

OnShareTargetActivated method 329

OnUnload event 293

Outlining menu 43

Overload Induction 94

P

Package.appxmanifest file 311, 321, 327

Page Inspector tool 221

parallel LINQ. *See* PLINQ

PE files 53

PerfMon

- about 104, 105
- working 106-108

performance 174

performance boosters

- in ASP.NET web applications 175-181

PLINQ 154

pointerDown animation 292

Portable Executable files. *See* PE files

PostAsync method 334

Post method 134

Prefetcher technology 182

PresentationCore.dll 227

PresentationFramework.dll 226

preventDefault method 284

private assembly

- about 60
- sharing 67-69

process 131 127

processAnchorClick method 283

ProcessRequestAsync method 213

ProgressChanged event 138

project template 29

PropertyChanged event 242

pruning 94

public assemblies 60

push notifications

- implementing, in Windows 8 style tile application 341-346

R

RangeValidator 182

ReaderWriterLock

about 146

for non-blocking synchronization constructs
146, 147

ready event 206

ready method 206

Reflector

about 163

assembly, disassembling 87, 88

option, using in Visual Studio 45-49

URL, for downloading 87

Relative Virtual Address. *See* **RVA**

ReportCompleted method 329

ReportProgress method 138

ReportStarted method 329

requestValidationMode attribute 222

RequiredFieldValidator 182

ResourceDictionary 225

Resume method 127

RibbonApplicationMenuItem 271

RibbonApplicationSplitMenuItem 271

RibbonButton

about 270

properties 270

RibbonButton, properties

ToolTipDescription 270

ToolTipImageSource 270

ToolTipTitle 270

RibbonComboBox 271

ribbon controls

shortcut keys 270

RibbonGallery

about 271

using, in ribbon 271, 272

RibbonMenuButton 271

RibbonMenuItem 271

RibbonSplitButton 271

RibbonSplitMenuItem 271

RibbonTooltip

for Ribbon-based controls 269

RibbonTooltip control 269

Ribbon User Interface

using, in WPF 260-269

RoutedEvents 234

RTTI address 102

runtime

configuration, modifying at 79

Run-time type information. *See* **RTTI address**

RVA 56

S

satellite assemblies 60

Section page 279

Section tag 192

semaphores 141

Send method 134

sequence diagram 25

serialization 78

services

about 330

working with 330-338

Settings charm, Windows 8 style tiles

application 310, 311

ShareOperation object 329

sharing 325

shortcut keys

for ribbon controls 270

show() method 205

sideloading 321

signaling construct 140

Signal() method 131

Silverlight 190

Sleep method 127

small object heap (SOH) 98

SmartAssembly

about 95

URL 95

used, for obfuscation 95, 96

Smart Tags

about 45

using, in Visual Studio 45-49

SmartTask dialog 215

sockets, HTML5 197-199

Solution Explorer

about 14, 175

exploring 15

images, previewing 16

working 15

- SOS (Son of Strike)** 99
- source applications** 329
- SpinLock**
 - about 145
 - used, for locking 145, 146
- Split App** 279
- stack** 97, 127
- StandardStyles section, XAML window** 302
- Start method** 127
- statically-typed model binding** 185-188
- storage files, Windows 8 style tiles application** 306-310
- suspended** 349
- Suspend method** 127
- SynchronizationContext class** 134, 135
- Syndication API**
 - about 330
 - used, for handling feeds in Windows 8 style application 346, 347
- System.Int32 type** 53
- System.Web.Optimization namespace** 180

T

- target applications** 329
- task-based asynchronous HttpHandlers**
 - working with 211-214
- task-based asynchronous HttpModules**
 - working with 211-214
- Task-based Asynchronous Pattern (TPL)** 127
- Task-based Parallelism Library (TPL)** 148
- Task List option** 25, 26
- task parallelism data flows**
 - about 165
 - working with 166-169
- tasks**
 - exceptions, handling 152
- TaskScheduler**
 - creating 153
- TDF blocks**
 - configuration options 169, 170
- templates**
 - about 29
 - creating 37, 38
 - extending 29-36
- Theme Animations** 305

- Theme Transitions** 304
- thread**
 - Mutex, determining 143, 144
- Thread Affinity** 134
- threading** 126-130
- ThreadLocal** 111-113
- Thread Locking**
 - about 139
 - working with 141, 142
- ThreadPool** 127
- ThreadPool.QueueUserWorkItem** 127
- threads** 125, 131
- threads, methods**
 - Abort 127
 - Join 127
 - Resume 127
 - Sleep 127
 - Start 127
 - Suspend 127
- thread synchronization**
 - about 139, 140
 - types 140
 - working with 141, 142
- thread synchronization, types**
 - locking constructs 140
 - non-blocking synchronization constructs 140
 - signaling construct 140
 - simple blocking methods 140
- ToggleButton** 225
- Toolbox**
 - using 44
- TPL**
 - without concurrency 153
- Twitter** 326

U

- UI** 223
- unmanaged heap** 97
- unmanaged layer, WPF** 227
- unobtrusive validation** 182, 183
- UpdateRetrieveTile method** 336
- User32** 227
- Usings**
 - organizing 42

V

validation 182

VB 7

VBC compiler 53

VB.NET 7

versions, Visual Studio

URL 9

Visual Studio Express 9

Visual Studio Premium 9

Visual Studio Professional 9

Visual Studio Ultimate 9

video tag 195

VIEW 234

VIEWMODEL 235

VirtualizingStackPanel 231

VisualStateManager class 303

Visual Studio

assembly, configuring 94, 95

Code Highlighting feature, using 22, 23

code snippets, using 39-42

command switches 13, 14

evolution 8

Refactor option, using 45-49

Smart Tags, using 45-49

templates, extending 29-36

used, for creating memory dump files 114

Visual Studio editors

enhancements 214-221

Visual Studio Express 9

Visual Studio IDE

components 8-13

files, previewing 21

Visual Studio Premium 9

Visual Studio Professional 9

Visual Studio Ultimate 9

Visual Trees 233

W

W3C 199

WaitHandle class

types 131

WaitHandle class, types

AutoResetEvent 131

CountdownEvent 131

ManualResetEvent 131

WaitHandle method 144

WeakEvent pattern

using, in WPF 272-274

weak references

about 108

types 109

weak references, types

large 109

short 109

Web 189

Web Authentication Broker API

used, for authenticating web service 338-341

WebAuthenticationBroker.AuthenticateAsync

method 341

WebAuthenticationBroker class 338

web.config file 178

WebGL 174

WebInvoke method 331

web service

authenticating, Web Authentication Broker API

used 338-341

web workers, HTML5 196, 197

wincontrol property 289

Windbg 104

windows

docking, inside IDE workspace 20

Windows 275

Windows 8 device

picture, taking from camera 311

Windows 8 environment

app enabling, for app sharing 326-329

Windows 8 Pro 276

Windows 8 style application

AtomPub API, used, for handling feeds 346, 347

Syndication API, used, for handling feeds 346, 347

Windows 8 style tiles application

animation, enabling with WinJS 291, 292

animation, implementing 304, 305

background transfers of data, performing 348-354

building, CSS used 278-290

building, C# used 297-302

building, HTML5 used 278-290

building, JavaScript used 278-290

- building, XAML used 297-302
- launching 320, 321
- layouts 302-304
- push notifications, implementing 341-346
- Settings charm 310, 311
- splash screen 320, 321
- storage files 306-310
- Theme Animations 305
- Theme Transitions 304
- WindowsBase.dll 227**
- WindowsCodecs.dll 227**
- Windows Presentation Foundation (WPF) 156**
- WinJS**
 - about 278
 - animation, enabling within Windows 8 style tiles application 291, 292
 - event life cycle, working 292, 293
 - library, writing for 294-296
- WinJS.UI.Animation namespace 291**
- WinRT**
 - about 276
 - application life cycle 312-320
 - supported languages 277, 278
- workspace area, of IDE**
 - working 18-20
- WPF**
 - about 223, 226
 - binding capabilities 225
 - built-in support, for animation 224
 - built-in support, for graphics 224
 - control template, redefining 224
 - enhancements, in .NET 4.5 226-232
 - managed layer 226
 - new property system 225
 - resource-based approach, for every control 225
 - Ribbon User Interface, using 260-269
 - styles, redefining 224
 - WeakEvent pattern, using 272-274
- WPFCorporateProject.zip file 32**
- WriteThreadName method 126**

X

XAML

- about 232, 297
- used, for building Windows 8 style tiles application 297-302

XAML window

- App section 302
- Frame section 302
- StandardStyles section 302



Thank you for buying Visual Studio 2012 and .NET 4.5 Expert Development Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.PacktPub.com.

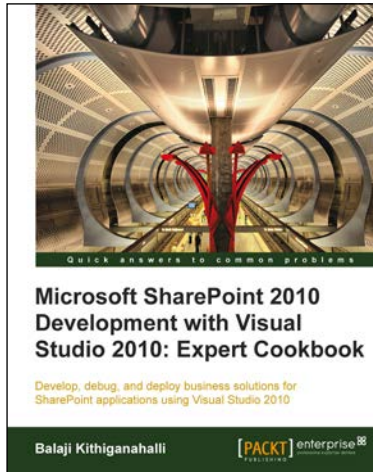
About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

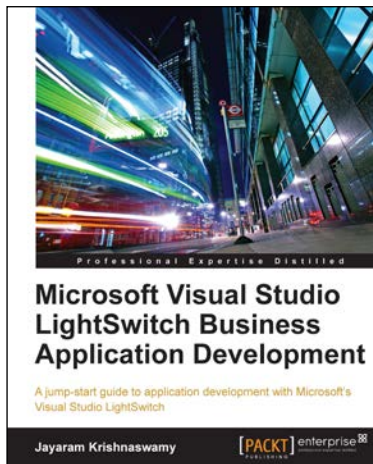


Microsoft SharePoint 2010 Development with Visual Studio 2010 Expert Cookbook

ISBN: 978-1-849684-58-3 Paperback: 296 pages

Develop, debug, and deploy business solutions for
SharePoint applications using Visual Studio 2010

1. Create applications using the latest client object model and create custom web services for your SharePoint environment with this book and ebook.
2. Full of illustrations, diagrams and key points for debugging and deploying your solutions securely to the SharePoint environment.
3. Recipes with step-by-step instructions with detailed explanation on how each recipe works and working code examples.



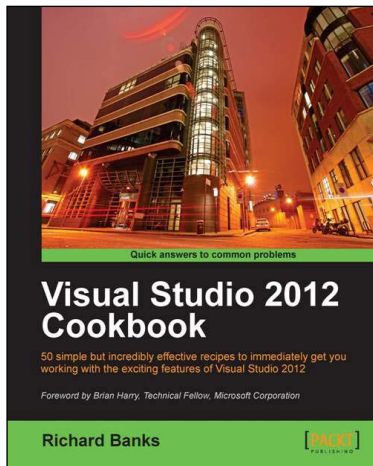
Microsoft Visual Studio LightSwitch Business Application Development

ISBN: 978-1-849682-86-2 Paperback: 384 pages

A jump-start guide to application development with
Microsoft's Visual Studio LightSwitch

1. A hands-on guide, packed with screenshots and step-by-step instructions and relevant background information—making it easy to build your own application with this book and ebook.
2. Easily connect to various data sources with practical examples and easy-to-follow instructions.
3. Create entities and screens both from scratch and using built-in templates.

Please check www.PacktPub.com for information on our titles



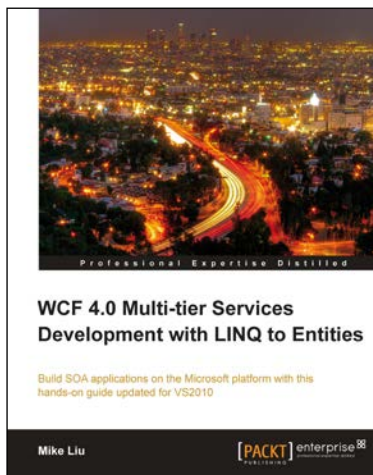
Visual Studio 2012 Cookbook

ISBN: 978-1-849686-52-5

Paperback: 272 pages

50 simple but incredibly effective recipes to immediately get you working with the exciting features of Visual Studio 2012

1. Take advantage of all of the new features of Visual Studio 2012, no matter what your programming language specialty is!
2. Get to grips with Windows 8 Store App development, .NET 4.5, asynchronous coding and new team development changes in this book and e-book.
3. A concise and practical First Look Cookbook to immediately get you coding with Visual Studio 2012.



LINQ to Entities

ISBN: 978-1-849681-14-8

Paperback: 348 pages

Build SOA applications on the Microsoft platform with this hands-on guide updated for VS2010

1. Master WCF and LINQ to Entities concepts by completing practical examples and applying them to your real-world assignments.
2. The first and only book to combine WCF and LINQ to Entities in a multi-tier real-world WCF service.
3. Ideal for beginners who want to build scalable, powerful, easy-to-maintain WCF services.
4. Rich with example code, clear explanations, interesting examples, and practical advice – a truly hands-on book for C++ and C# developers.

Please check www.PacktPub.com for information on our titles