C# 5 First Look

Write ultra responsive applications using the new asynchronous features of C#

Joel Martinez

[PACKT] PUBLISHING enterprise professional expertise distilled

C# 5 First Look

Write ultra responsive applications using the new asynchronous features of C#

Joel Martinez

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

C# 5 First Look

First published: December 2012

Production Reference: 1171212

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

ISBN 978-1-84968-676-1

www.packtpub.com

Cover Image by Faiz Fattohi (faizfattohi@gmail.com)

Credits

Author
Joel Martinez

Reviewers
Joydip Kanjilal
Asher De Vuyst

Acquisition Editor
Kevin Colaco

Commissioning Editor
Yogesh Dalvi

Technical Editors
Prasanna Joglekar
Vrinda Amberkar

Copy Editors
Laxmi Subramanian
Brandt D'Mello

Project Coordinator
Michelle Quadros

Proofreader
Kevin McGowan

Indexer
Hemangini Bari

Graphics
Aditi Gajjar

Production Coordinator
Melwyn D'sa

Cover Work
Melwyn D'sa

About the Author

Joel Martinez has most recently been focusing on mobile app development (Android, iOS, Windows Phone/RT). In the past, he founded the Orlando .NET User Group (ONETUG), worked at a few startups, made a few games, and was a Microsoft XNA MVP for a few years. You can find him on twitter at `@joelmartinez`, or on the Web at `http://codecube.net`.

He also co-authored the books *ASP.NET Development with Macromedia Dreamweaver MX*, *Peachpit Press*, *2002* and *Dreamweaver MX 2004 Magic*, *New Riders*, *2003*.

I would like to thank first and foremost my family (Tabbitha, Layla, and Ashton) for supporting (and tolerating) me during the writing of this book; everything I do is for you guys, I Love You! Thanks to my mom and dad, Ilu and Ramon, for being great parents and raising us right. My brother, Alex, for being someone I could always look up to, growing up. Gary, Charmayne, Alex (ermagherd am dern!), and Alyssa, you guys rock! To Igor and Howard, thank you for creating an environment where I can do fulfilling work with a great team, I'm very glad to be a part of the family. And finally, to the wonderful team at Packt Publishing, Yogesh and Michelle, it was a pleasure working with you.

About the Reviewers

Joydip Kanjilal is a Microsoft Most Valuable Professional in ASP.NET. He is also a speaker and author of several books and articles. He has over 14 years of industry experience in IT with more than 8 years in Microsoft .NET and its related technologies. He was selected as MSDN Featured Developer of the Fortnight (India) a number of times and also as Community Credit Winner at `www.community-credit.com` several times. Joydip has authored the following books:

- *Visual Studio 2010 Six in One, Wrox Publishing*
- *ASP.NET 4.0 Programming, McGraw-Hill Publishing*
- *Entity Framework Tutorial, Packt Publishing*
- *Pro Sync Framework, APRESS*
- *Sams Teach Yourself ASP.NET Ajax in 24 Hours, Sams Publishing*
- *ASP.NET Data Presentation Controls Essentials, Packt Publishing*

He has authored more than 200 articles for some of the most reputable sites such as `www.msdn.microsoft.com`, `www.asptoday.com`, `www.devx.com`, `www.ddj.com`, `www.aspalliance.com`, `www.aspnetpro.com`, `www.sql-server-performance.com`, and `www.sswug.com`. A number of these articles have been selected at `www.asp.net`—Microsoft's official site on ASP.NET.

He is currently working as an independent software consultant and author.
He has years of experience in designing and architecting solutions for various domains. His technical strengths include C, C++, VC++, Java, C#, Microsoft .NET, Ajax, WCF, REST, SOA, Design Patterns, SQL Server, Operating Systems, and Computer Architecture.

He blogs at `http://aspadvice.com/blogs/joydip` and spends most of this time writing books and articles. When not at work, he spends his time with his family, playing chess, and watching cricket and soccer.

You can see his MVP profile at `https://mvp.support.microsoft.com/default.aspx/profile/joydip`.

I am thankful to my family, friends, and Jini in particular, for inspiring me to take up this challenge. I enjoyed working on this book.

Asher De Vuyst is an American Software Engineer, Entrepreneur, proud father, and husband. He has designed software for the DOD, DOJ, investment banks, Disney, and several startups. Asher holds B.S. and M.S. degrees in computer engineering from the University of Central Florida and an M.B.A. from Rollins College. He is a member of the IEEE, ACM, and Tau Beta Pi.

Thanks to my family and friends for their patience and support.

---

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

C# is a wonderfully expressive and powerful language that lets you focus on your application rather than low-level boilerplate. Over the last decade, the C# compiler has evolved to include many features from dynamic and functional languages, all while remaining statically typed. Most recently, it has tackled the proliferation of concurrent hardware with new asynchronous programming features.

This book will help you get up to speed on the latest version of C#. After setting up your development environment, you will go on a tour of all the latest features of the language including the Task Parallel Framework, Dynamic Language Runtime, TPL Data Flow, and finally asynchronous programming with async and await.

What this book covers

Chapter 1, *Getting Started with C#*, gives a brief introduction to the birth of C#, and getting your development environment set up to compile C# 5. We will cover installation of the compiler and framework, along with all of the major editors such as Visual Studio and MonoDevelop.

Chapter 2, *Evolution of C#*, shows us how the C# language has grown and matured. With each release, new features were introduced that made programming in C# easier and more expressive.

Chapter 3, *Asynchrony in Action*, discusses asynchronous programming with a major focus on the 5.0 release. Starting with the Task Parallel Library (TPL), and culminating with the new async and await keywords that were newly introduced in this version of C#, you now have the ability to easily write responsive and scalable applications.

Chapter 4, *Creating a Windows Store App*, is about Windows 8 introducing a new application type, running on the WinRT framework, which lets you create applications that run on both x86 and ARM architectures. In this chapter, we explore the creation of a simple application that connects to the Internet to download and display images from Flickr.

Chapter 5, *Mobile Web App*, shows you how you have the ability to create very complex and compelling experiences for your users, with ASP.NET MVC and HTML 5. The world is going mobile and it is increasingly important that the Web supports mobile clients. You will learn how to build a mobile-optimized web application that uses HTML 5's geolocation APIs to connect users in real time.

Chapter 6, *Cross-platform Development*, shows you how, as a C# developer, you have the ability to target non-Microsoft platforms with the Mono Framework. In this chapter, you will learn how to create a utility application for Mac OS, using MonoMac and MonoDevelop. The ability to use C# could translate into a compelling opportunity, if you are able to share much of your code across platforms.

What you need for this book

In order to test and compile all of the examples in this book, you will need to install .NET 4.5 Framework, which is supported on all versions of Windows from Windows Vista and up which you can find at:

`http://www.microsoft.com/en-us/download/details.aspx?id=30653`

In order to compile and test the Windows Store and ASP.NET MVC projects (*Chapter 4*, *Creating a Windows Store App* and *Chapter 5*, *Mobile Web App* respectively), you will need to install a version of Visual Studio 2012 (`http://www.microsoft.com/visualstudio`). Additionally, the Windows Store project requires that you run Visual Studio 2012 on Windows 8.

The final project of the book in *Chapter 6*, *Cross-platform Development* is to create a Mac OS application using MonoMac (`http://www.mono-project.com/MonoMac`), and MonoDevelop (`http://monodevelop.com`). You must develop this on a Mac.

Who this book is for

This book is for developers who want to learn about the latest version of C#. It is assumed that you have basic programming knowledge. Experience with prior versions of C# or .NET Framework would be helpful, but not mandatory.

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: "By default, `csc` will output an executable file."

A block of code is set as follows:

```
using System;

namespace program
{
    class MainClass
    {
        static void Main (string[] args)
        {
            Console.WriteLine("Hello, World");
        }
    }
}
```

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

```
PS ~> $env:Path += ";C:\Windows\Microsoft.NET\Framework\v4.0.30319"
```

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: "Create a new project by clicking on **File | New Project…**.".

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 through 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/support`, 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 to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

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

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

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

Questions

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

1
Getting Started with C#

In this chapter, we will talk about the general state of the industry at the time when C# was first introduced, and some of the reasons that it is a great language. By the end of the chapter, you will have a fully working development environment that is ready to go through all of the examples in this book.

Origins

As every comic book super hero has an origin story, so does every professional in every walk of life. Sharing origin stories with your colleagues is great because it can serve as a point of reflection about how things were in the past, how they have evolved, and where they may go in the future. My own personal story originated in high school in the late nineties, watching my brother, who was five years my elder and in college, learning C++. With a few arcane instructions, complex programs came to life and were ready for action. I was fascinated.

This first glimpse of power was just the beginning. Around the same time, a friend of mine in class started working on a game, again written in C++, in the style of the NES game, *The Legend of Zelda*. Although I had briefly peeked at old QBasic games such as *Gorillas* in the past, I was amazed at the quality that he was able to achieve in his small demo. I decided to start learning how to program in earnest, and seeing as everyone I knew was using C++, that was the default choice for my first programming language.

The first program I ever wrote was a very simple financial budgeting program. Having just recently started working at my first job in high school, I was keenly aware of the new responsibilities involved in the management of money, and so I wanted to write a program to help me better manage my funds. First, it asked for the amount of my paycheck in dollars, and then for a list of bills that I had to pay.

After a few basic calculations, it gave me a report of how much disposable income would remain after my responsibilities were taken care of. As far as programs go, it was not the most complex piece of software, but it helped me learn the basics, such as loops, conditional statements, storage of an indeterminate list of items, and performing aggregate operations on an array.

It was a great personal triumph, but after some initial exploration with C++, I found myself hitting a bit of a brick wall. C++ was difficult to fully grasp as someone brand new to programming (and in high school). Not only did I have to learn about the basics of software, but I had to be constantly aware of the memory that I was using. Eventually, I discovered the tools of web development which were, to me at the time, much simpler to understand. I had moved from one end of the complexity spectrum to the other.

Much of the software landscape at that time was dominated by computer languages that sat in one of three camps: low level system languages such as C++, which offered the most in terms of performance and flexibility, but were also difficult and complex to master; interpreted languages such as JavaScript and VBScript, whose instructions were evaluated at runtime, were very easy to use and learn, but could not match the performance of low level languages; and finally a group of languages that come somewhere in the middle.

This middle of the road, which encompassed languages such as Java and Visual Basic, offered some of the best of both worlds, along with the worst of both worlds. In these languages you have a **garbage collector**, which means that when you create an object you do not have to explicitly release the used memory when you are done. They are also compiled to an intermediate language (for example, p-code for VB, and byte code for Java) which are then executed in a Virtual Machine running natively on the target platform. Because this intermediate language is similar to machine code, it is able to execute much faster than the purely interpreted languages. This performance, however, was still not really a match for a properly tuned C++ program, so Java and Visual Basic programs were often regarded as slow languages in comparison with C++.

Despite some of these drawbacks, the benefits of having a managed memory environment were evident to Microsoft. Because the programmer did not have to worry about complex concepts such as pointers and manual memory management, programs could be written faster and with fewer bugs. **Rapid Application Development** (**RAD** for short) seemed to be the future direction for Microsoft's platforms.

In the late nineties, they developed a version of the Java Virtual Machine, which by many accounts was faster than some of the other implementations available on the market. Unfortunately due to their inclusion of some proprietary extensions, and the fact that they did not completely implement the Java 1.1 standard, they ran into some legal troubles in 1997. This resulted ultimately in Microsoft discontinuing the development on their implementation of Java, and ultimately removing it from their platform in 2001.

Although it is impossible to know if what happened next was a direct result of the legal action against the Microsoft Java Virtual Machine, what we do know is that in 1999 Microsoft started working on a new programming language, which was named **Cool** (**C-like Object Oriented Language**).

C# is born

And then it happened; in 2000, Microsoft announced that they were working on a new programming language. The language which was originally called Cool, was unveiled at the Professional Developers Conference 2000 in Orlando, FL as **C#**. Some of the highlights of this new language are:

- It is based on the syntax of the C family of programming languages, so the syntax was very familiar for anyone who had experience with C++, Java, or JavaScript.
- Memory management of C# is similar to that of Java and Visual Basic, with a very powerful garbage collector. This meant that the users could focus on the content of their application, rather than worrying about boilerplate memory management code.
- The C# compiler along with a static type system means that certain classes of bugs can be caught at compile time, rather than having to deal with them at runtime as you would in JavaScript. This is a **Just-In-Time** compiler, which means that the code would be compiled to a native executable at runtime, and optimized for the operating system that is executing the code. Performance is an important goal of the new platform.
- This language has a strong and extensive **base class library**, which means that many pieces of functionality would be built right into the framework. Aside from some industry standard libraries such as Boost, there were not very many common C/C++ libraries, which resulted in people often rewriting common functionality. Java, on the other hand, had a great many libraries, but they were written by a diverse set of developers, which meant that consistency in functionality and style was a problem.

- It also has interoperability with other languages that worked on the **Common Language Runtime** (**CLR**). So a single program could use functionality written in different languages, thus using each language for what it was best at.
- Microsoft submitted the specification to the ISO working group. This opened the door to a vibrant open source community around the framework, because it meant that there would always be a standard to work against. A popular open source implementation of the .NET Framework and C# called **Mono** lets you run your code on different platforms.

Although none of the elements described in this list were particularly new, C# aimed to take the best aspects of programming languages that came before, and incorporate them, namely the strength and power of C++, the simplicity of JavaScript, and the ease of hosting of VBScript/ASP, among other things.

People coming from ANY language (C, C++, or Java) could be productive in C# with little effort. C# found the sweet spot where productivity, features, and the learning curve all intersected.

Over the next decade, the language would go on to evolve a very attractive set of features that make it easier and faster to write great programs. Now in its fifth iteration, the C# language has become more expressive and powerful with features, such as **Language Integrated Queries** (**LINQ**), **Task Parallel Library** (**TPL**), a **Dynamic Language Runtime** (**DLR**), and asynchronous programming features. What's more, with the Mono framework, you can not only target Windows, but also every other mainstream platform such as Linux, Mac OS, Android, iOS, and even game consoles such as the Playstation Vita.

Whether you have been writing C# for the last decade, or are just picking it up now, this book will take you through all of the features of the latest version 5.0. We will also explore the evolution and history of C# so that you can understand why certain features developed the way they did, and how you can use them to their full potential.

Before we begin though, we need to configure your computer to be able to compile all of the samples. This chapter will guide you through installing everything you need to go through every example in this book.

The tools

Whenever you approach a new programming language, or a tool, there are several questions that you can ask yourself in order to quickly become proficient in that environment, such as:

- How do you build a program, or otherwise prepare it for deployment?
- How do you debug a program? Quickly figuring out what the problem is, and where it is when there is one. This is just as important as writing the program in the first place.

In the following sections, we will review several tools that are available to you in order to get a development environment up and running on your local machine. These options vary across a number of different licensing terms and cost structures. No matter your situation or preferences, you will able to get a development environment up and running and you will be able to answer the previous questions by the end of the chapter.

Visual Studio

Microsoft provides the de facto compiler and development environment for the C# language. Although the compiler is available as a command-line executable since the first release of the .NET Framework, most developers will stay within the confines of Visual Studio, which is Microsoft's **Integrated Development Environment** (**IDE**).

Full versions of Visual Studio

Microsoft's full commercial offerings of Visual Studio come in several different versions, each with a cumulative number of features as you move up the ladder.

- **Professional**: This is the base commercial package. It allows you to build all available projects, in all available languages. In the context of C#, some of the project types available are ASP.NET WebForms, ASP.NET MVC, Windows 8 App, Windows Phone, Silverlight, Library, Console, along with a robust testing framework.
- **Premium**: In this version, all professional features are included, in addition to the code metrics, expanded testing tools, architecture diagramming, lab management, and project management features.
- **Ultimate**: This version includes code clone analysis, more testing tools (including Microsoft Fakes), and IntelliTrace, in addition to all the features of the previous levels.

Check out these versions at `http://www.microsoft.com/visualstudio/11/en-us/products/visualstudio`.

Licensing

There are several different options for licensing the full version of Visual Studio.

- **MSDN Subscription**: The Microsoft Developer Network provides a subscription service where you can pay an annual fee to gain access to versions of Visual Studio. Additionally, you can get an MSDN Subscription as part of Microsoft's MVP program, which rewards the active community members in the development community. You can find more information about purchasing an MSDN Subscription at `https://msdn.microsoft.com/en-us/subscriptions/buy/buy.aspx`.
- **BizSpark**: If you are creating a startup, Microsoft offers the BizSpark program to give you access to Microsoft software (including Visual Studio) at no cost for three years. After your graduation date, you keep the licenses that you've downloaded over the course of the program, and get discounts on MSDN Subscriptions, in addition to other alumni benefits. BizSpark is a great option for any entrepreneur that wants to use the Microsoft technology stack. Find out if you qualify for the BizSpark program at `http://www.microsoft.com/bizspark`.
- **DreamSpark**: Students can enroll in the DreamSpark program, which lets you download Visual Studio Professional (in addition to other applications and servers). As long as you are a student in a valid academic institution, you will have access to everything you need to develop applications using C#.Students. Sign up today at `https://www.dreamspark.com/`.
- **Individual and Volume licensing**: If none of the previous options for the commercial version of Visual Studio are appropriate, then you can always purchase licenses directly from Microsoft or various resellers at `http://www.microsoft.com/visualstudio/en-us/buy/small-midsize-business`.

Express

The **Visual Studio Express** product line is a nearly fully featured version of Visual Studio that is free of cost. Anyone can download these products and begin learning and developing at no charge.

The available versions are as follows:

- **Visual Studio Express 2012 for Windows 8**: This is for creating Metro style applications for Windows 8
- **Visual Studio Express 2012 for Windows Phone**: This lets you write programs for Microsoft's Windows Phone devices

- **Visual Studio Express 2012 for Web**: All web applications can be built using this version of Visual Studio, from ASP.NET (forms and MVC), to Azure hosted projects
- **Visual Studio Express 2012 for Desktop**: Applications that target the *classic* Windows 8 Desktop environment can be built with this version.

It's a common misconception that Visual Studio Express may only be used for non-commercial projects, but this is not the case. You are entirely free to develop and release a commercial product while still adhering to the EULA. The only limitations are technical, as follows:

- Express versions of Visual Studio are limited by vertical stack, meaning you have to install a separate product for each project type that is supported (Web, desktop, phone, and so on). This is hardly a huge limitation though, and would only be a burden in the most complex of solutions.
- There are no plugins. There are many productivity enhancing plugins that are available for the full version of Visual Studio, so for some users this exclusion can be a big deal. However, the good news is that one of the most popular plugins in recent memory, **NuGet**, is now being shipped with all versions of Visual Studio 2012. NuGet helps you manage your project's library dependencies. You can browse through the NuGet catalog and add open source third-party libraries, in addition to libraries from Microsoft.

The express versions of Visual Studio can be downloaded from `http://www.microsoft.com/visualstudio/11/en-us/products/express`.

Using Visual Studio

Regardless of which version of Visual Studio you decide to use, getting started is very simple once the product has been installed. The following are the steps:

1. Launch Visual Studio, or if you are using Express, launch Visual Studio Express for Desktop.
2. Create a new project by clicking on **File** | **New Project…**.
3. Choose **Console Application** from **Installed** | **Templates** | **Visual C#**.
4. Give the project a name such as `program`, and click on **OK**.
5. Add a line of code in the `Main` method as follows:

```
Console.WriteLine("Hello World");
```

6. Run the program by choosing **Debug** | **Run without Debugger**.

You will see the expected **Hello World** output and you are now ready to start using Visual Studio.

Command-line compiler

If you prefer to work at a lower level than with an IDE like Visual Studio, you can always opt to simply use the command-line compiler directly. Microsoft provides everything you need to compile C# code entirely for free by downloading and installing the .NET 4.5 Redistributable package from `http://www.microsoft.com/en-us/download/details.aspx?id=8483`.

Once that's downloaded and installed, you can find the compiler at `C:\windows\microsoft.net\Framework\v4.0.30319\csc.exe`, assuming you maintain all of the default installation options:

Note that the .NET 4.5 version of the .NET Framework will actually replace the 4.0 framework if you have that installed. That's why the path mentioned previously shows as `v4.0.30319`. You won't be the first person confused by versions in the .NET Framework.

A small tip that will make working with the command-line compiler much easier is to simply add it to the environment's `Path` variable. If you're using PowerShell (which I highly encourage), you can easily do so by running the following command:

```
PS ~> $env:Path += ";C:\Windows\Microsoft.NET\Framework\v4.0.30319"
```

That makes it so you can just type `csc` instead of the whole path. Usage of the command-line compiler is very simple, take the following class:

```
using System;

namespace program
{
    class MainClass
    {
        static void Main (string[] args)
        {
            Console.WriteLine("Hello, World");
        }
    }
}
```

Save this class as a file named `program.cs` using your favorite text editor. Once saved, you can compile it from the command line using the following command:

```
PS ~\book\code\ch1> csc .\ch1_hello.cs
```

This will produce an executable file named `ch1_hello.exe`, which when executed, will produce a familiar greeting as follows:

```
PS ~\book\code\ch1> .\ch1_hello.exe
Hello, World
```

By default, `csc` will output an executable file. However, you can also produce libraries using the target argument. Consider the following class:

```
using System;

namespace program
{
    public class Greeter
    {
        public void Greet(string name)
        {
            Console.WriteLine("Hello, " + name);
        }
    }
}
```

This class encapsulates the functionality of the previous program, and even makes it reusable by letting you define the name to be greeted. Although this is a somewhat trite example, the point is to show how to create a `.dll` file that you can use from multiple programs.

```
PS ~\dev\book\code\ch1> csc /target:library .\ch1_greeter.cs
```

An assembly named `ch1_greeter.dll` will be generated, which you can then use from a slightly modified version of the previous program as follows:

```
using System;

namespace program
{
    class MainClass
    {
        static void Main (string[] args)
        {
            Greeter greeter = new Greeter();
            greeter.Greet("Componentized World");
        }
    }
}
```

If you try to compile the previous program just as you did before, the compiler will rightly complain about not knowing anything about the `Greeter` class as follows:

```
PS ~\book\code\ch1> csc .\ch1_greeter_program.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17626
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

ch1_greeter_program.cs(9,13): error CS0246: The type or
        namespace name 'Greeter' could not be found (are you
        missing a using directive or an assembly reference?)
ch1_greeter_program.cs(9,35): error CS0246: The type or
        namespace name 'Greeter' could not be found (are you
        missing a using directive or an assembly reference?)
```

Any time you have an error in your programs, it will be shown in the output, along with information about the file it was found in, and the line, so you can find it easily. In order for this to work, you will have to tell the compiler to use the `ch1_greeter.dll` file that you created using the `/r:` argument as follows:

```
PS ~\book\code\ch1> csc /r:ch1_greeter.dll .\ch1_greeter_program.cs
```

And now when you run the resulting `ch1_greeter_program.exe` program, you will see the output say, **Hello, Componentized World**.

Though most developers will not use the command-line compiler directly these days, it is good to know that it is available and also how to use it, especially if you have to support advanced scenarios such as merging multiple modules into a single assembly.

SharpDevelop

When you launch SharpDevelop, the tagline on the loading screen, **The Open Source .NET IDE**, is a concise description. since the very early days of the .NET Framework, it provided developers a free option for writing C# before Microsoft shipped the Express versions. Since that time, it has continued to mature, and add features, and as of version 4.2, SharpDevelop supports targeting the .NET 4.5, and more specifically, compilation and debugging of C# 5.0. Although Visual Studio Express is a compelling option, the lack of source control plugins can be a deal breaker for some users. Thankfully, SharpDevelop will gladly let you integrate with a source control server in the IDE. Additionally, some of the more niche project types such as creating Windows Services (one of the few project types not supported by Express) are fully supported with SharpDevelop.

Projects use the same format (`.sln`, `.csproj`) as Visual Studio, so project portability is high. You can usually take a project written in Visual Studio and open it in SharpDevelop.

Download the application from `http://www.icsharpcode.net/OpenSource/SD/`.

Installation is straightforward, and you can verify correct installation by creating the following sample program:

1. Start SharpDevelop.
2. Create a new project by clicking on **File** | **New** | **Solution**.
3. Choose **Console Application** from **C#** | **Windows Application**.
4. Give the project a name such as `program`, and click on **Create**.
5. Right-click on the project node in the **Projects** window, and choose the **Properties** menu item; check the **Compiling** tab to see if the **Target Framework** says **.NET Framework 4.0 Client Profile**.
6. If it does, then simply click on the **Change** button, select **.NET Framework 4.5** in the **Change Target Framework** drop-down menu, and finally click on the **Convert** button.
7. Run the program by choosing **Debug** | **Run without Debugger**.

You will see the expected **Hello World** output.

MonoDevelop

The **Mono** framework is an open source version of the Common Language Runtime and C#. It has had over a decade of active development, and as a result, is very mature and stable. There are versions of Mono for just about any platform you might be interested in developing for Windows, OS X, Unix/Linux, Android, iOS, PlayStation Vita, Wii, and Xbox 360.

MonoDevelop is based on SharpDevelop, but was forked some time ago to specifically act as a development environment for Mono that would run on multiple platforms. It runs on Windows, OS X, Ubuntu, Debian, SLE, and openSUSE; so, as a developer, you can truly choose what platform you want to work on.

You can get started by installing the Mono Development Kit 2.11 or higher from `http://www.go-mono.com/mono-downloads/download.html`.

Once you have installed that for your platform, you can go ahead and install the latest version of MonoDevelop from `http://monodevelop.com/`.

Using the C# 5.0 compiler is but a few short steps away:

1. Start MonoDevelop.
2. Create a new project by clicking on **File | New | Solution…**.
3. Choose **Console Application** from the **C#** menu.
4. Give the project a name such as `program`, and click on **Forward**, then on **OK**.
5. Right-click on the project node in the **Solution** window, and choose the **Options** menu item. Now go to **Build | General** to see if the **Target Framework** says **Mono / .NET 4.0**.
6. If it does, then simply choose **.NET Framework 4.5** from the dropdown and click on the **OK** button.
7. Run the program by choosing **Run | Start without Debugging**.

If all goes well, you will see a terminal window (if running on OS X, for example) with the **Hello World** text.

Summary

When it was introduced, C# came as a breath of fresh air at just the right time. It is a modern object oriented language, which takes the best qualities of many that came before it. Low-level power with just-in-time compilation, the simplicity of a garbage collected environment, and the flexibility of runtime that allows for easy interoperability with other languages, not to mention a great base class library, thriving open source community, and the ability to target multiple platforms make C# a compelling option.

In this chapter, we discussed setting up your development environment and downloading all of the relevant tools and runtimes:

- Visual Studio, both commercial and free options.
- Command line, useful for plugging directly into automated tools that use shell commands.
- SharpDevelop, an open source alternative to Visual Studio.
- MonoDevelop, the official IDE of the open source implementation of the .NET Framework and C#. This allows you to target multiple platforms.

Once you have chosen a preferred development environment, and followed the steps detailed in this chapter, you will be ready to go through the rest of the book and all of the samples contained therein.

In the next chapter, we will talk about the evolution of the language, which, in turn, will help you understand the feature that was introduced along the way and contributes to where we stand today with C# 5.0.

2
Evolution of C#

In this chapter, we look back at the entire history of C#, leading up to the latest version. We will not be able to cover everything, but we will touch on the major features, especially the ones which are historically relevant. Each release brought unique features that would serve as the building blocks for innovations in versions yet to come.

C# 1.0 – in the beginning

When C# was introduced, Microsoft wanted to take the best features of many other languages and runtimes. It was an object oriented language, with a runtime that actively manages memory for you. Some of the features of this language (and framework) are:

- Object oriented
- Managed memory
- Rich base class library
- Common Language Runtime
- Type safety

Regardless of your technology background, there was something in C# that you could relate to.

Runtime

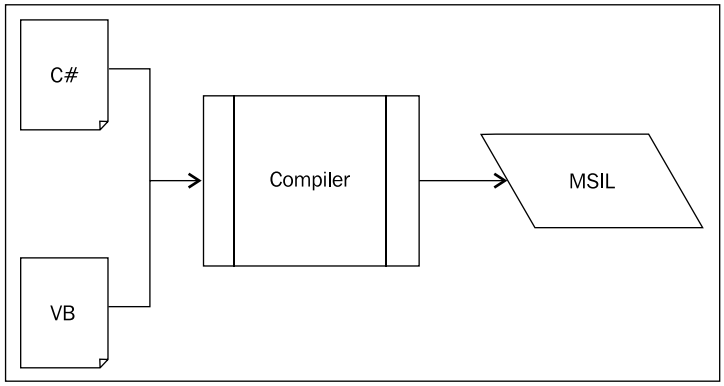
It is impossible to talk about C#, without first talking about the runtime, called the **Common Language Runtime** (**CLR**). One of the major underpinnings of the CLR was the ability to interoperate with multiple languages, which meant that you would be able to write your program in a number of different languages, and it would run on the same runtime. This interoperability was accomplished by agreeing to a common set of data types, referred to as the **common type system**.

Before the .NET Framework, there was really no clear mechanism for different languages to talk to each other. A string in one environment may not match the concept of a string in another language and would they be null terminated? Are they encoded in ASCII? How is that number represented? There was simply no way to know, because each language did its own thing. Of course people tried to come up with solutions to this problem.

In the Windows world, the most well-known solution was to use the **Component Object Model** (**COM**), which used a type library to describe the types contained therein. By exporting this type library, a program could talk to other processes that may or may not have been written using another technology, because you were sharing details about how to communicate. However this was not without complexity, as anyone who wrote COM libraries using Visual Basic 6 could tell you. Not only was the process generally somewhat opaque because the tool abstracted out the underlying technology, but deployment was a nightmare. There was even a well-known phrase for working with it, DLL Hell.

The .NET Framework was, in part, a response to this problem. It introduces the common type system into the picture, which are rules that every language running on the CLR needs to adhere to, including common data types such as strings and numeric types, the way object inheritance works, and type visibility.

For maximum flexibility, instead of compiling directly to native binary code, an intermediate representation of program code is used as the actual binary image, which is distributed and executed, called **MSIL**. This MSIL is then compiled the first time you run the program, so that optimizations can be put in place for the specific processor architecture that the program is being run on (the **Just-In-Time** (**JIT**) compiler). This means that a program that runs on the server and on a desktop could have different performance characteristics based on the hardware. In the past, you would have had to compile two different versions.



Another benefit of inherent, multilingual support, is that it served as a migration strategy. A number of different languages came out at the same time as C#. Companies that had an existing codebase in various languages could easily convert their program to a .NET friendly version, which was CLR compatible, and subsequently use it from other .NET languages such as C#. Some of the languages include the following:

- **VB.NET**: This is the natural successor to the popular Visual Basic 6.
- **J#**: This is a version of the Java language for .NET.
- **Managed extensions for C++**: With a flag, and a few new keywords and syntax, you could take an existing C++ application, and compile it to a version compatible with the CLR.

While a number of these languages shipped, and were pitched as fully supported, these days the only ones that really remain of the original languages that shipped are VB.Net and C#, and to a lesser degree, C++/CLI. Many new languages such as F#, IronPython, and IronRuby have sprung up on the CLR over the years, and they remain active in development, with vibrant communities.

Memory management

The Common Language Runtime provides a **garbage collector**, which means that memory will be collected automatically when an object is no longer referenced by other objects. This was not a new concept of course; many languages such as JavaScript and Visual Basic support garbage collection. Unmanaged languages, on the other hand, let you manually allocate memory on the heap if you so choose. And although this ability gives you way more power when it comes to the kinds of low-level solutions you can implement, it also gives you more opportunities to make mistakes.

The following are the two kinds of data types that the CLR allows for:

- **Value types**: These data types are created using the `struct` keyword
- **Reference types**: These data types are created using the `class` keyword

Every primitive data type in C#, such as `int` and `float`, is `struct`, while every class is a reference type. There are some semantics around how these types are allocated internally (stack versus heap), but for day-to-day use those differences are usually not important.

You can of course create your own custom types of both kinds. For example, the following is a simple value type:

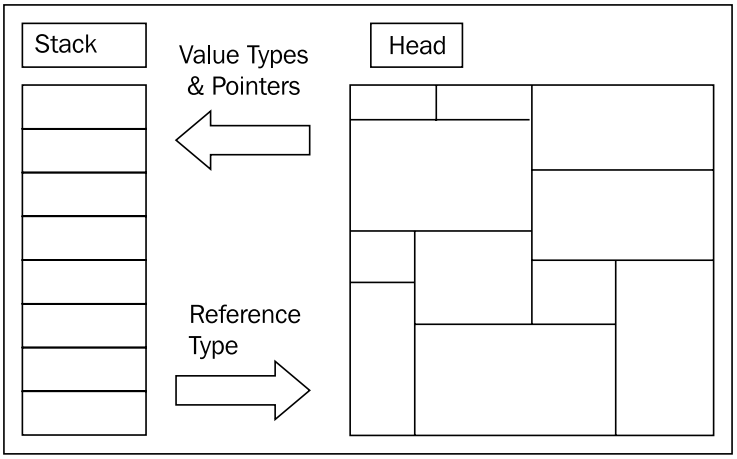
```
public struct Person
{
    public int Age;
    public string Name;
}
```

By changing a single keyword, you can change this object to a reference type as follows:

```
public class Person
{
    public int Age;
    public string Name;
}
```

There are two major differences between `struct` instances and `class` instances. Firstly, a `struct` instance cannot inherit, or be inherited from. The `class` instances, however, are the primary vehicles for creating object oriented hierarchies.

Secondly, `class` instances participate in the garbage collection process, while `struct` instances do not, at least not directly. Many discussions on the Internet tend to generalize the memory allocation strategy of value types as being allocated on the stack, while reference types are allocated on the heap (see the following diagram), but that is not the whole story:



There is generally some truth to this, because when you instantiate `class` in a method, it will always go on the heap, while creating a value type such as an `int` instance will go on the stack. But if the value type is wrapped in a reference type, for example, when the value type is a field in a `class` instance, then it will be allocated on the heap along with the rest of the class data.

Syntax features

The name C# is a cheeky reference to the C language, just as C++ was C (plus some stuff), C# too is largely similar in syntax to C and C++, though with some obvious changes. Unlike C, C# is an object oriented language, but with a few interesting features that make it easier and more efficient to write than other object oriented languages.

One example of this is the property getters and setters. In other languages, if you want to control how you expose access and modifications to the data of a `class` instance, you would have to make the field private so an outsider could not change it directly. Then you would make two methods, one prefixed with `get` to retrieve the value, and one prefixed with `set` to the value. In C#, the compiler takes care of generating those method pairs for you. Look at the following code:

```
private int _value;

public int Value
{
    get { return _value; }
    set { _value = value; }
}
```

Another interesting innovation is how C# provides first class event support. In other languages such as Java, they approximated events by, for example, having a method called `setOnClickListener(OnClickListener listener)`. To use it, you have to define a new class that implements the `OnClickListener` interface and pass it in. This technique definitely works, but can be kind of verbose. With C#, you can define what is called a **delegate** to represent a method as a proper, self-contained object as follows:

```
public delegate void MyClickDelegate(string value);
```

This `delegate` can then be used as an event on a class as follows:

```
public class MyClass
{
    public event MyClickDelegate OnClick;
}
```

To register for notification when the event is raised, you can just create the delegate and use the += syntax to add it to the delegate list as follows:

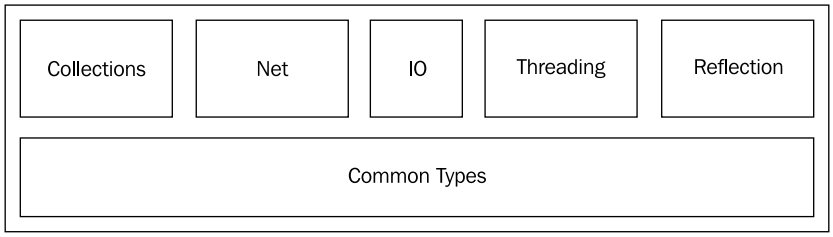
```
public void Initialize()
{
    MyClass obj = new MyClass();
    obj.OnClick += new MyClickDelegate(obj_OnClick);
}

void obj_OnClick(string value)
{
    // react to the event
}
```

The language will automatically add the delegate to a list of delegates, which will be notified whenever the event is raised. In Java, that kind of behavior would have to be implemented manually.

There were many other interesting syntactic features when C# was launched, such as the way exceptions worked, the using statement, and others. But in the interest of brevity, let's move on.

Base Class Library

By default, C# comes with a rich and vast framework called the **Base Class Library** (**BCL**). The BCL provides a wide array of functionality as shown in the next diagram:



The diagram shows a few of the namespaces that are included in the base class library (emphasis on a few). While there are a large number of other namespaces, these are a few of the most important ones, which provide the infrastructure for many of the features and libraries that have been released by Microsoft and third parties.

One of the data structure types you discover when learning how to program is the one that deals with collection of information. Typically, you learn to write most of the programs and algorithms using arrays. With an array though, you have to know the size of the collection ahead of time. The `System.Collections` namespace comes

with a set of collection of data structures that make it easy to handle an unknown amount of data.

In the very first program I ever wrote (described briefly in the previous chapter), I used an array that was pre-allocated. To keep things simple, the array was allocated with an arbitrarily large number of elements so that I would not run out of spaces in the array. Of course, that would never work in a non-trivial program written professionally because you will either run out of space if you encounter a larger than expected set of data, or it will be wasteful of memory. Here, we can use one of the most basic collection types, the `ArrayList` collection, to overcome that problem, as follows:

```
// first, collect the paycheck amount
Console.WriteLine("How much were you paid? ");
string input = Console.ReadLine();
float paycheckAmount = float.Parse(input);

// now, collect all of the bills
Console.WriteLine("What bills do you have to pay? ");
ArrayList bills = new ArrayList();
input = Console.ReadLine();
while (input != "")
{
    float billAmount = float.Parse(input);
    bills.Add(billAmount);
    input = Console.ReadLine();
}

// finally, summ the bills and do the final output
float totalBillAmount = 0;
for (int i = 0; i < bills.Count; i++)
{
    float billAmount = (float)bills[i];
    totalBillAmount += billAmount;
}

if (paycheckAmount > totalBillAmount)
{
    Console.WriteLine("You will have {0:c} left over after paying
bills", paycheckAmount - totalBillAmount);
}
else if (paycheckAmount < totalBillAmount)
{
    Console.WriteLine("Not enough money, you need to find an extra
{0:c}", totalBillAmount - paycheckAmount);
```

```
}
else
{
    Console.WriteLine("Just enough to cover bills");
}
```

As you can see, an instance of the `ArrayList` collection was created, but no size was specified. This is because collection types manage their sizes internally. This abstraction relieves you of the size responsibility so you can worry about bigger things.

Some of the other collection types that are available are as follows:

- `HashTable`: This type allows you to provide a lookup key and a value. It is often used to create very simple in-memory databases.
- `Stack`: This is a first in, last out data structure.
- `Queue`: This is a first in, first out data structure.

Looking at the concrete collection, classes do not really tell the whole story though. If you follow the inheritance chain, you will notice that every collection implements an interface called `IEnumerable`. This will come to be one of the most important interfaces in the whole language so getting familiar with it early on is important.

`IEnumerable`, and the sister class, `IEnumerator`, abstract the concept of enumeration over a collection of items. You will always see these interfaces used in tandem, and they are very simple. You can see this as follows:

```
namespace System.Collections
{
    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }

    public interface IEnumerator
    {
        object Current { get; }
        bool MoveNext();
        void Reset();
    }
}
```

At first glance, you may wonder why collections implement `IEnumerable`, which has a single method that returns an `IEnumerator`, rather than just implementing `IEnumerator` directly. The enumerator is responsible for enumerating through a collection. But there is a good reason for this. If the collection itself was the enumerator, then you would not be able to iterate over the same collection concurrently. So each call to `GetEnumerator()` will generally return a separate enumerator, though that is by no means a requirement.

Although the interface is very simple, it turns out that having this abstraction is very powerful. C# implements a nice shorthand syntax for iterating over a collection without having to do the regular `for` loop using an index variable that you have to pass in. This is explained in the following code:

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;

foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

The `foreach` syntax works because it is shorthand for the code the compiler will actually generate. It will generate code to interact with the enumerable behind the scenes. So the loop in the previous example will look like the compiled MSIL, as follows:

```
IEnumerator enumerator = numbers.GetEnumerator();

while (enumerator.MoveNext())
{
    int number = (int)enumerator.Current;
    Console.WriteLine(number);
}
```

Once again, we have an example of the C# compiler generating code, that is different from what you have actually written. This will be the key in the evolution of C# to make the common patterns of code that you write easier to express, allowing you to be more efficient and productive.

To some developers, C# was a cheap imitation of Java when it first came out. But to developers like me, it was a breath of fresh air, offering performance improvements over interpreted languages such as VBScript, extra safety and simplicity from languages such as C++, and more low level power than languages such as JavaScript.

C# 2.0

The first major update of the C# language, Runtime, and .NET Framework was a big one. This release focused on making the language more concise and easier to write.

Syntax updates

The first update added a small capability to the property syntax. In 1.0, if you wanted a read only property, your only choice was to exclude the setter, as follows:

```
private int _value;

public int Value
{
    get { return _value; }
}
```

All internal logic had to interact with the `_value` member directly. In many cases this was fine, except for cases where you needed to have some sort of logic governing when and how you were allowed to change that value. Or similarly, if you needed to raise an event, you would have to create a private method as follows:

```
private void SetValue(int value)
{
    if (_value < 5)
        _value = value;
}
```

Well no more in C# 2.0, as you can now create a private setter as follows:

```
private int _value;

public int Value
{
    get { return _value; }
    private set
    {
        if (_value < 5)
            _value = value;
    }
}
```

A small feature, but it increased consistency because separate getter and setter methods were one of the things that C# tried to get rid of from the first version.

Another interesting addition is that of **nullable** types. With value types, the compiler will not allow you to set them to a null value, however, you now have a new key character that you can use to signify a nullable value type as follows:

```
int? number = null;
if (number.HasValue)
{
    int actualValue = number.Value;
    Console.WriteLine(actualValue);
}
```

Just by adding the question mark, the value type is marked as nullable, and you can use the `.HasValue` and `.Value` properties to make decisions on what to do in the case of null.

Anonymous methods

Delegates are a great addition to C# over other languages. They are the building blocks of the event systems. One drawback, however, in the way they were implemented in C# 1.0 is that they make reading code a bit more difficult, because the code that executes when the event is raised is actually written elsewhere. Continuing the trend of code simplification, **anonymous methods** let you write the code inline. For example, given the following delegate definition:

```
public delegate void ProcessNameDelegate(string name);
```

You can create an instance of the delegate using an anonymous method as follows:

```
ProcessNameDelegate myDelegate = delegate(string name)
{
    Console.WriteLine("Processing Name = " + name);
};

myDelegate("Joel");
```

This code is inline, short, and easy to understand. It also allows you to use delegates much like first-class functions in other languages such as JavaScript. But it goes beyond simply being easier to read. If you wanted to pass a parameter to a delegate that did not accept a parameter in C# 1.0, you had to create a custom class to wrap both the method implementation and the stored value. This way, when the delegate is invoked (thus executing the target method), it has access to the value. Any early multi-threaded code was full of code like the following:

```
public class CustomThreadStarter
{
    private int value;
```

```
        public CustomThreadStarter(int val)
        {
            this.value = val;
        }

        public void Execute()
        {
            // do something with 'value'
        }
    }
```

This class accepts a value in the constructor and stores it in a private member. Then later when the delegate is invoked, the value can be used, as in this case using it to start a new thread. This is shown in the following code:

```
CustomThreadStarter starter = new CustomThreadStarter(55);
ThreadStart start = new ThreadStart(starter.Execute);
Thread thread = new Thread(start);
thread.Start();
```

With anonymous delegates, the compiler can step in and greatly simplify the usage pattern mentioned previously as follows:

```
int value = 55;
Thread thread = new Thread(delegate()
    {
        // do something with 'value'
        Console.WriteLine(value);
    });
thread.Start();
```

This might look simple, but there is some serious compiler magic going on here. The compiler has analyzed the code, realized that the anonymous method requires the `value` variable in the method body, and automatically generated a class similar to the `CustomThreadStarter` that we would have had to create in C# 1.0. The result is code that you can easily read because it is all there, right in context with the rest.

Partial classes

In C# 1.0, it was common practice to use code generators to automate things such as custom collections. When you wanted to add your own methods and properties to the generated code, you would generally have to inherit from the class, or in some cases, directly edit the generated file. This meant that you had to be very careful to avoid regenerating the code, or risk overwriting your custom logic. You will find a comment similar to the following one in many first generation tools:

```
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.3053
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </auto-generated>
```

C# 2.0 adds an additional keyword to your arsenal, `partial`. With **partial** classes, you can break up your classes among multiple files. To see this in action, create the following class:

```
// A.generated.cs
public partial class A
{
    public string Name;
}
```

This represents the automatically generated code. Notice that the file contains `.generated` in the filename; this is a convention that was adopted, though is not necessary for this to work, it is just that both files are part of the same project. Then in a separate file, you can include the rest of the implementation as follows:

```
// A.cs
public partial class A
{
    public int Age;
}
```

All members would then be available on the resulting type at runtime, as the compiler takes care to stitch the class together. You are free to regenerate the first file at will, without the risk of overwriting your changes.

Generics

The major feature addition of C# 2.0 is **generics**, which allows you to create classes that can be reused with multiple types of objects. In the past, this kind of programming could only be accomplished in two ways. You can use a common base class for the parameter, so that any object that inherits from that class can be passed in regardless of the concrete implementation. That works, sort of, but it becomes very limiting when you want to create a very general purpose data structure. The other method is really just a derivative of the first. Instead of using a base class of your own definition, go all the way up the inheritance tree and use `object` for your type parameter.

This works because all the types in .NET derive from `object`, so you can pass in anything. This is the method used by the original collection classes. But even this has problems, especially when it comes to passing in value types due to the effects of boxing. You also have to cast the type back out from object every single time.

Thankfully, all of these problems can be mitigated by using generics as follows:

```
public class Message<T>
{
    public T Value;
}
```

In this example, we have defined a **generic type parameter** called `T`. The actual name of the generic type parameter can be anything, `T` is just used as a convention. When you instantiate the `Message` class, you can specify the kind of object you want to store in the `Value` property using this syntax, as follows:

```
Message<int> message = new Message<int>();
message.Value = 3;
int variable = message.Value;
```

So you can assign an integer to the field without worrying about performance, because the value will not be boxed. You also do not have to cast it when you want to use it, as you would if using object.

Generics are super powerful, but they are not omnipotent. To highlight a key deficiency, we will go over one of the first things that just about every C# developer tried when 2.0 was first released—generic math. Developers of applications that are math heavy will likely be using a mathematical library for their domain. For example, game developers (or really, anyone doing anything that involves 2D or 3D spatial calculations) will always need a good `Vector` structure as follows:

```
public struct Vector
{
    public float X;
    public float Y;

    public void Add(Vector other)
    {
        this.X += other.X;
        this.Y += other.Y;
    }
}
```

But the problem is that it is using the `float` data type for calculations. If you wanted to generalize it and support other numeric types such as `int`, `double`, or `decimal`, what do you do? Upon first glance, you would think that you could use generics to support this scenario as follows:

```
public struct Vector<T>
{
    public T X;
    public T Y;

    public void Add(Vector<T> other)
    {
        this.X += other.X;
        this.Y += other.Y;
    }
}
```

Compiling this will result in an error, **Operator '+=' cannot be applied to operands of type 'T' and 'T'**. This is because, by default, only members from the `object` data type are available for the generic parameter, due to the fact that the compiler has no way of knowing what methods (and by extension, operations) are defined on the type you are using.

Thankfully, Microsoft anticipated this to some degree, and added something called **generic type constraints**. These constraints let you give the compiler a hint at what kind of types callers will be allowed to use, which in turn means that you can use the features that you constrain. For example, look at the following code:

```
public void WriteIt<T>(T list) where T : IEnumerable
{
    foreach (object item in list)
    {
        Console.WriteLine(item);
    }
}
```

Here, we have added a constraint that says that the type parameter `T` must be an `IEnumerable`. As a result, you can write the code and be safe in the knowledge that any caller that calls this method will only ever use a type that implements the `IEnumerable` interface as the type parameter. Some of the other parameter constraints you can use are as follows:

- `class`: This says that the type parameter must be a reference type.
- `struct`: This implies that only value types are allowed.

- `new()`: There must be a public constructor without parameters on this type. It will allow you to use syntax like `T value = new T()` to create new instances of the type parameter. Otherwise, the only thing you can do is something like `T value = default(T)`, which will return null for reference types, and zero for numeric primitives.
- `<name of interface>`: This limits the type parameters to use the interface mentioned here, as shown with `IEnumerable` mentioned previously.
- `<name of class>`: Any type used with this constraint must be of this type, or inherit from this type at some point in the inheritance chain.

Unfortunately, because numeric data structures are value types, they cannot inherit, and thus have no common type to use in a type constraint that will give you the mathematical operators needed to do math.

As a general rule of thumb, generics are most useful in "framework" style code, which is to say general infrastructure for your applications, or data structures such as collections. In fact, some great new collection types became available in C# 2.0.

Generic collections

Generics are perfect for collections because the collection itself doesn't really have to interact with the objects that it contains; it just needs a place to put them. So with a collection, there are no constraints on the type parameter. All of the new generic collections can be found in the namespace as follows:

```
using System.Collections.Generic;
```

As we discussed earlier, the most basic collection type that was in C# 1.0 was an `ArrayList` collection, which worked really well at the time. However, value types would be boxed as it used `object` as its payload type, and you had to cast the object out into your target object type every time you wanted to pull out a value. With generics, we now have `List<T>` as follows:

```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);

int value = list[1]; // returns 2;
```

The usage is practically identical to the `ArrayList` collection, but with the performance benefits of generics. Some of the other types available as generic classes are as follows:

- `Queue<T>`: This is the same as the non-generic `Queue`, **first in, first out** (**FIFO**).
- `Stack<T>`: There are no differences here from the non-generic version of the `Stack`, **last in, first out** (**LIFO**).
- `Dictionary<T, K>`: This takes the place of the `Hashtable` collection from C# 1.0. It uses two generic parameters for the key and value of each dictionary item. This means that you can use a key other than a string.

Iterator methods

Perhaps one of the more unique features to arrive in C# 2.0 was that of **iterator** methods. They are a way of having the compiler automatically generate a custom iteration over a sequence. That description is kind of abstract, admittedly, so the easiest way to explain it is with some code, as follows:

```
private static IEnumerable<string> GetStates()
{
    yield return "Orlando";
    yield return "New York";
    yield return "Atlanta";
    yield return "Los Angeles";
}
```

In the previous method, you see a method that returns `IEnumerable<string>`. In the method body, however, there are simply four consecutive lines of code that use the `yield` keyword. This tells the compiler to generate a custom enumerator that breaks up the method into each individual part between the yields, so that it is executed when a caller enumerates the returned value. This is shown in the following code:

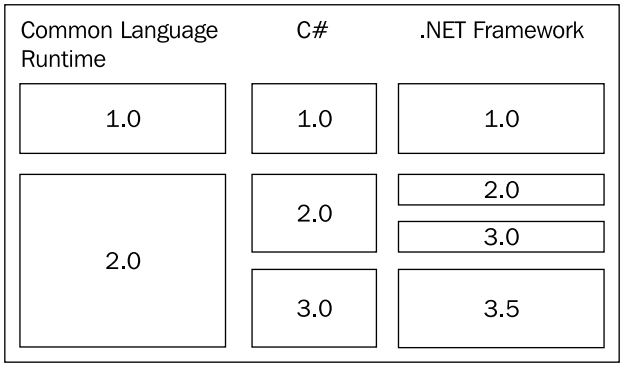
```
foreach (string state in GetStates())
{
    Console.WriteLine(state);
}
// outputs Orlando, New York, Atlanta, and Los Angeles
```

There are a lot of different ways to approach and use iterators, but the highlight here is how the C# compiler is getting smarter in this release. It is able to take your code and expand it. This lets you write code at a higher level of abstraction, which is an ongoing theme in the evolution of C#.

C# 3.0

If you thought C# 2.0 was a big update, the 3.0 release was even bigger! It is difficult to do justice to it in a single chapter (let alone part of a chapter). So we are going to focus on the main features, especially as it relates to the evolution of C#.

First though, we should talk about the difference between C#, the CLR, and the .NET Framework. Up until now, they all mostly had the same version (that is C# 2.0, CLR 2.0, and .NET Framework 2.0), however, they released an update to the .NET Framework (3.0) that had no language or CLR changes. Then with .NET 3.5, they released C# 3.0. The following diagram explains these differences:



Confusing, I know. Although both the C# language and the .NET Framework received an upgrade, the CLR remained unchanged. It is hard to believe, especially in light of all the new features, but it goes to show how forward thinking the developers of the CLR have been, and how well-engineered and extensible the C# language/compiler is that they were able to add new features without new runtime support.

Syntax updates

As usual, we will begin reviewing the syntactic changes of the language for this release. First are properties, which as you will remember are already an improvement over the old school getter and setter methods. In C# 3.0, the compiler can automatically generate the backing field for simple getters and setters as follows:

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

This feature alone cuts out many lines of code from classes that have many properties. Another nice feature introduced is that of **object initializers**. Take the following simple class:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

If you want to create an instance and initialize it, you would normally have to write code as follows:

```
Person person = new Person();
person.Name = "Layla";
person.Age = 11;
```

But with an object initializer, you can do this at the same time as object instantiation as follows:

```
Person person = new Person { Name = "Layla", Age = 11 };
```

The compiler will actually generate pretty much the same code as before, so there is no semantic difference. But you can write your code in a much more concise and easy-to-read manner. Collections get a similar treatment as you can now initialize arrays with the following syntax:

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

And dictionaries, which were notoriously verbose to initialize, can now be created very easily as follows:

```
Dictionary<string, int> states = new Dictionary<string,int>
{
    { "NY", 1 },
    { "FL", 2 },
    { "NJ", 3 }
};
```

Each of these improvements makes the very act of typing code easier and quicker. But it seems that the C# language designers were not content to stop there. Every time you instantiated a new variable, you were forced to write out the entire type name, when you start getting to complex generic types this can add a lot of extra characters to your program. Fear not! You do not even need to do that in C# 3.0! Look at the following code:

```
var num = 8;
var name = "Ashton";
var map = new Dictionary<string, int>();
```

As long as it is clear what type is being assigned on the right side of the equation, the compiler can take care of figuring out the type on the left side. Astute readers will no doubt recognize the `var` keyword from JavaScript. Though it looks similar, it is not the same at all. C# is still statically typed, which means that every variable must be known at compile time. The following code will not compile:

```
var num = 8;
num = "Tabbitha";
```

So, in effect, this is just a shortcut to help you type fewer characters, the compiler is just really good at inferring these things. In fact, that is not the only thing it can infer. If there is enough context, it can also infer generic type parameters. For example, consider the following simple generic method:

```
public string ConvertToString<T>(T value)
{
    return value.ToString();
}
```

When you call it, rather than stating the type in the call, the compiler can look at the type of class that is being passed in, and simply assume that this is the type that should be used for the type parameter as follows:

```
string s = ConvertToString(234);
```

At this point, I imagine someone on the C# language team said: "While we're making existing syntax optional, why not do away with the need for class definitions entirely!" And it would seem they did just that. If you need a data type to hold a few fields, you can declare it inline as follows:

```
var me = new { Name = "Joel", Age = 31 };
```

The compiler will automatically create a class that matches the type that you just created. There are a few limitations to this feature: you have to use the `var` keyword, and you cannot return an anonymous type from a method. Very useful when you are writing an algorithm and need a quick, yet complex data type.

All of these little syntax changes add up and make the C# language a pleasure to write in. They also are a lead in for the next big feature we are going to talk about.

LINQ

Language Integrated Query(**LINQ**) is the flagship feature of C# 3.0. It acknowledges the fact that much of a modern day program revolves around querying for data in one way or another. LINQ is a set of diverse features that gives the language first class support for querying data from a multitude of sources. It does so by providing a strong abstraction around the concept of querying, and then adding language support.

The C# language team started with the premise that SQL was already a great syntax for working with set-based data. But, unfortunately, it was not a part of the language; it required a different runtime, such as SQL Server, and only worked in that context. LINQ requires no such context switch, so you can simply get a reference to your data source, and query away.

Conceptually, there are the following, high level kind of operations that you can do with a set:

- **Filtering**: This is performed where you exclude items from a set based on some criteria
- **Aggregation**: This involves common aggregation actions such as grouping, and summation
- **Projection**: This is extracting or converting items from a set

The following is what a simple LINQ query looks like:

```
int[] numbers = { 1, 2, 3, 4, 5, 6 };

IEnumerable<int> query = from num in numbers
                         where num > 3
                         select num;

foreach (var num in query)
{
    Console.WriteLine(num);
}
// outputs 4, 5, and 6
```

It looks like SQL, kind of. There have been many questions over the years over why the syntax does not start with the select statement like it does in SQL, but the reason comes down to tooling. When you start typing, they want you to be able to get IntelliSense when typing every part of the query. By starting with the 'from', you are essentially telling the compiler what type will be used in the rest of the query, which means it can give you type-time support.

One of the interesting things about LINQ is that it works for any `IEnumerable`. Think about that for a second, every single collection in your program is now easily searchable. And that is not all, you can aggregate and shape the output as well. For example, say you wanted to get a count of cities in each state as follows:

```
var cities = new[]
{
    new { City="Orlando", State="FL" },
    new { City="Miami", State="FL" },
    new { City="New York", State="NY" },
    new { City="Allendale", State="NJ" }
};

var query = from city in cities
            group city by city.State into state
            select new { Name = state.Key, Cities = state };

foreach (var state in query)
{
    Console.WriteLine("{0} has {1} cities in this collection", state.
Name, state.Cities.Count());
}
```

This query uses the group by clause to group the values by a common key, in this case by state. The final output is also a new anonymous type that has two properties, the name, and the collection of cities in that state. Running this program will output this for Florida as **FL has 2 cities in this collection**.

So far in these examples, we have been using what is called **query syntax**. This is nice because it is very familiar to those who know SQL. However, just as with SQL, more complex queries can sometimes get rather verbose and complicated to read. There is another way to write LINQ queries that, for some, can be much easier to read, and perhaps even slightly more flexible called the **LINQ method syntax**, it is built upon another new feature of the language.

Extension methods

Normally, the only way of extending functionality of a type is to inherit from the class and add the features to the subtype. All users have to use the new type to get the benefits of that new type. However, this may not always be an option, for example, if you are using a third-party library with value types (as you cannot inherit from a value type). Let us say we have the following `struct` in a third-party library, where we do not have access to modify the source code:

```
public struct Point
{
    public float X;
    public float Y;
}
```

With extension methods, you have the capability to add new methods to this type as follows:

```
public static class PointExtensions
{
    public static void Add(this Point value, Point other)
    {
        value.X += other.X;
        value.Y += other.Y;
    }
}
```

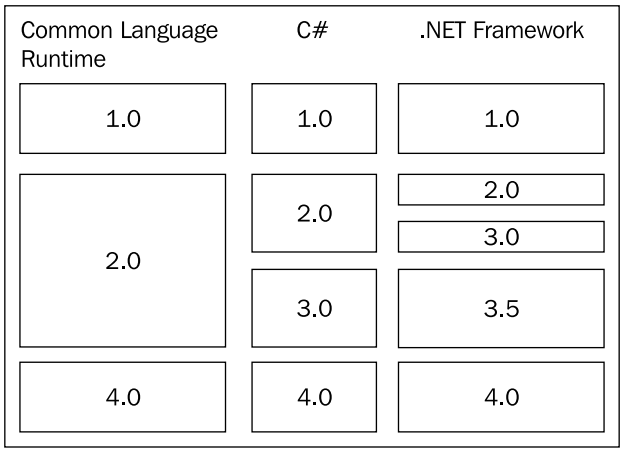
Extension methods must be placed in a public static class. The method itself will be static, and will use the `this` keyword on the first parameter to signify the type to attach to. Using the previous method looks like the method has always been a part of the type as follows:

```
var point = new Point { X = 28.5381f, Y = 81.3794f };
var other = new Point { X = -2.6809f, Y = -1.1011f };

point.Add(other);
Console.WriteLine("{0}, {1}", point.X, point.Y);
// outputs "25.8572, 80.2783"
```

You can add the extension methods to any type, whether value type, or reference type. Also interfaces and sealed classes can be extended. If you look at all of the changes in C# 3.0, you will notice that you are now writing less code because the compiler is generating more and more of it behind the scenes for you. The result is code that looks similar to some of the other dynamic languages such as JavaScript.

C# 4.0

With the fourth iteration of the language, Microsoft tried to simplify the versioning confusion it created over the previous few releases by incrementing the version of every component to 4.0.



C# 4.0 brings more dynamic functionality into the language and continues the work of making C# a very powerful, yet agile language. Some of the features added are primarily to make interoperation with native platform code easier. Things such as covariance, contra variance, and optional parameters, simplify the process of doing things, calling the interop assemblies for interacting with Microsoft Word, for example. All in all, not very earth-shaking stuff, at least for your average day-to-day developer.

However, with a new keyword that was added, `dynamic`, C# takes a step closer to becoming a very dynamic language; or at the very minimum, inheriting many of the qualities of dynamic languages. Remember when generics were introduced, if you had a bare type parameter (that is, with no type constraints), it was treated as an object. The compiler had no additional information about what kind of methods and properties the type had access to at runtime, and as such you could only interact with it as an object.

In C# 4.0, you now have a way of writing code that can bind to the correct properties and methods at runtime. The following is a simple example:

```
dynamic o = GetAString() ;

string s = o.Substring(2, 3);
```

If you are migrating a project from an earlier version of the framework, make sure you add a reference to `Microsoft.CSharp.dll`, when using dynamic programming. You will receive a compilation error if this is not present.

In this hypothetical scenario, you have a method that returns a `string`. The variable that is receiving the return value of the `GetAString()` method is marked with the `dynamic` keyword. This means that every property and method that you call on that object will be dynamically evaluated at runtime. This lets C# easily interop with dynamic languages, such as IronPython and IronRuby, in addition to your own custom dynamic types.

Does this mean that C# is no longer statically typed? No, quite the opposite; C# is still statically typed, just that in this case you have told the compiler to handle this code differently. It does this by rewriting your dynamic code to use the **Dynamic Language Runtime** (**DLR**), which actually compiles out expression trees of your code that are evaluated at runtime.

You can easily create your own dynamic objects by inheriting from the built-in class called `DynamicObject` as follows:

```
public class Bag : DynamicObject
{
    private Dictionary<string, object> members = new
Dictionary<string, object>();

    public override IEnumerable<string> GetDynamicMemberNames()
    {
        return members.Keys;
    }

    public override bool TryGetMember(GetMemberBinder binder, out
object result)
    {
        return members.TryGetValue(binder.Name, out result);
    }

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

In this simple example, we inherit from `DynamicObject` and override a few methods to get and set the member value. These values are stored internally in a dictionary so that you can pull out the correct value when the DLR asks for it. Using this class is very reminiscent of how flexible objects are in JavaScript. Look at the following code:

```
dynamic bag = new Bag();

bag.Name = "Joel";
bag.Age = 31;
bag.CalcDoubleAge = new Func<int>(() => bag.Age * 2);

Console.WriteLine(bag.CalcDoubleAge());
```

If you need to store a new value, simply set the property. And if you want to define a new method, you can use a delegate as the value for the member. Of course, you must realize that this will not be as fast as having a regular statically typed class, every value must be looked up at runtime, and because values are stored as objects internally, any value type will be boxed. But sometimes those drawbacks are completely acceptable, especially when it can simplify your code.

Summary

For me, it has been an amazing journey, watching C# evolve from the very first version until today. Each subsequent release was more powerful than the previous, and there was a very solid theme of code simplification throughout. The compiler itself has gotten better and better at generating code on your behalf, so that you can implement very powerful features in your programs without having the cognitive burden of verbosely implementing the infrastructure (generics, iterators, LINQ, and the DLR)

In this chapter we looked at some of the major features that were introduced in each version of C#

- **C# 1.0**: Memory Management, Base Class Library, and syntax features such as properties and events.
- **C# 2.0**: Generics, iterator methods, partial classes, anonymous methods, and syntactic updates such as visibility modifiers on properties and nullable types.
- **C# 3.0**: **Language Integrated Query** (**LINQ**), extension methods, automatic properties, object initializers, type inference (`var`), and anonymous types.
- **C# 4.0**: The **Dynamic Language Runtime** (**DLR**), and co- and contra variance

Now, we move to the latest release, C# 5.0.Iliquamet quae volor aut ium ea dolore doleseq uibusam, quiasped utem atet etur sus.

3
Asynchrony in Action

We are going to explore features which are new to C# in the 5.0 release. Notably, most of them are related to built-in asynchrony features added to the language, which allow you to easily use the hardware running your software to its full potential. We will also discuss **Task Parallel Library** (**TPL**), which introduces primitives for asynchronous programming, C# 5.0 language support for easy asynchrony, TPL DataFlow, higher-level abstractions for agent-based asynchronous programming, and framework improvements that take advantage of the new asynchrony features, such as improvements to the I/O APIs.

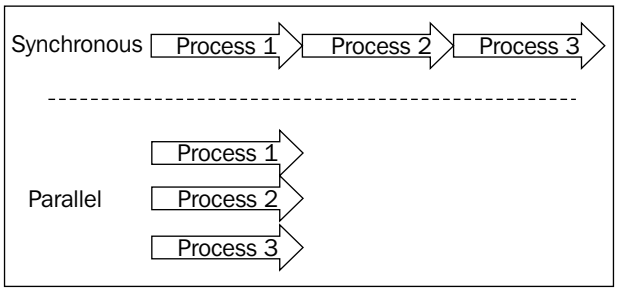
All things considered, the latest release of the .NET Framework is enormous, and the concepts introduced in this chapter will serve as reference for the material covered in the rest of the book.

Asynchrony

When we talk about C# 5.0, the primary topic of conversation is the new asynchronous programming features. What does **asynchrony** mean? Well, it can mean a few different things, but in our context, it is simply the opposite of synchronous. When you break up the execution of a program into asynchronous blocks, you gain the ability to execute them side-by-side, in parallel.

> *Pit of Success: in stark contrast to a summit, a peak, or a journey across a desert to find victory through many trials and surprises, we want our customers to simply fall into winning practices by using our platform and frameworks. To the extent that we make it easy to get into trouble we fail.* –*Rico Mariani*

Unfortunately, building asynchronous software has not always been easy, but with C# 5.0 you will see how easy it can be. As you can see in the following diagram, executing multiple actions concurrently can bring various positive qualities to your programs:



Parallel execution can bring performance improvements to the execution of a program. The best way to put this into context is by way of an example, an example that has been experienced all too often in the world of desktop software.

Let's say you have an application that you are developing, and this software should fulfill the following requirements:

1. When the user clicks on a button, initiate a call to a web service.
2. Upon completion of the web service call, store the results into a database.
3. Finally, bind the results and display them to the user.

There are a number of problems with the naïve way of implementing this solution. The first is that many developers write code in such a way that the user interface will be completely unresponsive while we are waiting to receive the results of these web service calls. Then, once the results finally arrive, we continue to make the user wait while we store the results in a database, an operation that the user does not care about in this case.

The primary vehicle for mitigating these kinds of problems in the past has been writing multithreaded code. This is of course nothing new, as multi-threaded hardware has been around for many years, along with software capabilities to take advantage of this hardware. Most of the programming languages did not provide a very good abstraction layer on top of this hardware, often letting (or requiring) you program directly against the hardware threads.

Thankfully, Microsoft introduced a new library to simplify the task of writing highly concurrent programs, which is explained in the next section.

Task Parallel Library

The **Task Parallel Library** (**TPL**) was introduced in .NET 4.0 (along with C# 4.0). We did not cover it in *Chapter 2*, *Evolution of C#*, for several reasons. Firstly, it is a huge topic and could not have been examined properly in such a small space. Secondly, it is highly relevant to the new asynchrony features in C# 5.0, so much so that they are the literal foundation upon which the new features are built. So, in this section, we will cover the basics of the TPL, along with some of the background information about how and why it works.

TPL introduces a new type, the `Task` type, which abstracts away the concept of *something that must be done* into an object. At first glance, you might think that this abstraction already exists in the form of the `Thread` class. While there are some similarities between `Task` and `Thread`, the implementations have quite different implications.

With a `Thread` class, you can program directly against the lowest level of parallelism supported by the operating system, as shown in the following code:

```
Thread thread = new Thread(new ThreadStart(() =>
{
Thread.Sleep(1000);
Console.WriteLine("Hello, from the Thread");
    }));
thread.Start();

Console.WriteLine("Hello, from the main thread");
thread.Join();
```

In the previous example, we create a new `Thread` class, which when started will sleep for a second and then write out the text **Hello, from the Thread**. After we call `thread.Start()`, the code on the main thread immediately continues and writes **Hello, from the main thread**. After a second, we see the text from the background thread printed to the screen.

In one sense, this example of using the `Thread` class shows how easy it is to branch off the execution to a background thread, while allowing execution of the main thread to continue, unimpeded. However, the problem with using the `Thread` class as your "concurrency primitive" is that the class itself is an indication of the implementation, which is to say, an operating system thread will be created. As far as abstractions go, it is not really an abstraction at all; your code must both manage the lifecycle of the thread, while at the same time dealing with the task the thread is executing.

If you have multiple tasks to execute, spawning multiple threads can be disastrous, because the operating system can only spawn a finite number of them. For performance intensive applications, a thread should be considered a heavyweight resource, which means you should avoid using too many of them, and keep them alive for as long as possible. As you might imagine, the designers of the .NET Framework did not simply leave you to program against this without any help. The early versions of the frameworks had a mechanism to deal with this in the form of the `ThreadPool`, which lets you queue up a unit of work, and have the thread pool manage the lifecycle of a pool of threads. When a thread becomes available, your work item is then executed. The following is a simple example of using the thread pool:

```
int[] numbers = { 1, 2, 3, 4 };

foreach (var number in numbers)
{
ThreadPool.QueueUserWorkItem(new WaitCallback(o =>
        {
Thread.Sleep(500);
            string tabs = new String('\t', (int)o);
Console.WriteLine("{0}processing #{1}", tabs, o);
        }), number);
}
```

This sample simulates multiple tasks, which should be executed in parallel. We start with an array of numbers, and for each number we want to queue a work item that will sleep for half a second, and then write to the console. This works much better than trying to manage multiple threads yourself because the pool will take care of spawning more threads if there is more work. When the configured limit of concurrent threads is reached, it will hold work items until a thread becomes available to process it. This is all work that you would have done yourself if you were using threads directly.

However, the thread pool is not without its complications. First, it offers no way of synchronizing on completion of the work item. If you want to be notified when a job is completed, you have to code the notification yourself, whether by raising an event, or using a thread synchronization primitive, such as `ManualResetEvent`. You also have to be careful not to queue too many work items, or you may run into system limitations with the size of the thread pool.

With the TPL, we now have a concurrency primitive called `Task`. Consider the following code:

```
Task task = Task.Factory.StartNew(() =>
    {
Thread.Sleep(1000);
```

```
Console.WriteLine("Hello, from the Task");
    });

Console.WriteLine("Hello, from the main thread");

task.Wait();
```

Upon first glance, the code looks very similar to the sample using `Thread`, but they are very different. One big difference is that with `Task`, you are not committing to an implementation. The TPL uses some very interesting algorithms behind the scenes to manage the workload and system resources, and in fact, allows you customize those algorithms through the use of custom schedulers and synchronization contexts. This allows you to control the parallel execution of your programs with a high degree of control.

Dealing with multiple tasks, as we did with the thread pool, is also easier because each task has synchronization features built-in. To demonstrate how simple it is to quickly parallelize an arbitrary number of tasks, we start with the same array of integers, as shown in the previous thread pool example:

```
int[] numbers = { 1, 2, 3, 4 };
```

Because `Task` can be thought of as a primitive type that represents an asynchronous task, we can think of it as data. This means that we can use things such as Linq to project the numbers array to a list of tasks as follows:

```
var tasks = numbers.Select(number =>
Task.Factory.StartNew(() =>
    {
Thread.Sleep(500);
        string tabs = new String('\t', number);
Console.WriteLine("{0}processing #{1}", tabs, number);
    }));
```

And finally, if we wanted to wait until all of the tasks were done before continuing on, we could easily do that by calling the following method:

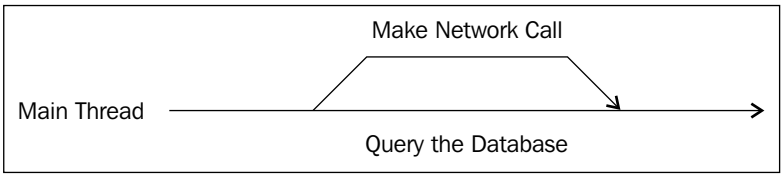
```
Task.WaitAll(tasks.ToArray());
```

Once the code reaches this method, it will wait until every task in the array completes before continuing on. This level of control is very convenient, especially when you consider that, in the past, you would have had to depend on a number of different synchronization techniques to achieve the very same result that was accomplished in just a few lines of TPL code.

With the usage patterns that we have discussed so far, there is still a big disconnect between the process that spawns a task, and the child process. It is very easy to pass values into a background task, but the tricky part comes when you want to retrieve a value and then do something with it. Consider the following requirements:

1. Make a network call to retrieve some data.
2. Query the database for some configuration data.
3. Process the results of the network data, along with the configuration data.

The following diagram shows the logic:



Both the network call and query to the database can be done in parallel. With what we have learned so far about tasks, this is not a problem. However, acting on the results of those tasks would be slightly more complex, if it were not for the fact that the TPL provides support for exactly that scenario.

There is an additional kind of `Task`, which is especially useful in cases like this called `Task<T>`. This generic version of a task expects the running task to ultimately return a value, whenever it is finished. Clients of the task can access the value through the `.Result` property of the task. When you call that property, it will return immediately if the task is completed and the result is available. If the task is not done, however, it will block execution in the current thread until it is.

Using this kind of task, which promises you a result, you can write your programs such that you can plan for and initiate the parallelism that is required, and handle the response in a very logical manner. Look at the following code:

```
varwebTask = Task.Factory.StartNew(() =>
    {
WebClient client = new WebClient();
        return client.DownloadString("http://bing.com");
    });

vardbTask = Task.Factory.StartNew(() =>
    {
        // do a lengthy database query
        return new
        {
```

```
WriteToConsole=true
        };
    });

if (dbTask.Result.WriteToConsole)
{
Console.WriteLine(webTask.Result);
}
else
{
ProcessWebResult(webTask.Result);
}
```

In the previous example, we have two tasks, the `webTask`, and `dbTask`, which will execute at the same time. The `webTask` is simply downloading the HTML from `http://bing.com`. Accessing things over the Internet can be notoriously flaky due to the dynamic nature of accessing the network so you never know how long that is going to take. With the `dbTask` task, we are simulating accessing a database to return some stored settings. Although in this simple example we are just returning a static anonymous type, database access will usually access a different server over the network; again, this is an I/O bound task just like downloading something over the Internet.

Rather than waiting for both of them to execute like we did with `Task.WaitAll`, we can simply access the `.Result` property of the task. If the task is done, the result will be returned and execution can continue, and if not, the program will simply wait until it is.

This ability to write your code without having to manually deal with task synchronization is great because the fewer concepts a programmer has to keep in his/her head, the more resources he/she can devote to the program.

If you are curious about where this concept of a task that returns a value comes from, you can look for resources pertaining to "Futures", and "Promises" at:

`http://en.wikipedia.org/wiki/Promise_%28programming%29`

At the simplest level, this is a construct that "promises" to give you a result in the "future", which is exactly what `Task<T>` does.

Task composability

Having a proper abstraction for asynchronous tasks makes it easier to coordinate multiple asynchronous activities. Once the first task has been initiated, the TPL allows you to compose a number of tasks together into a cohesive whole using what are called **continuations**. Look at the following code:

```
Task<string> task = Task.Factory.StartNew(() =>
{
WebClient client = new WebClient();
    return client.DownloadString("http://bing.com");
});

task.ContinueWith(webTask =>
    {
Console.WriteLine(webTask.Result);
    });
```

Every task object has the `.ContinueWith` method, which lets you chain another task to it. This continuation task will begin execution once the first task is done. Unlike the previous example, where we relied on the `.Result` method to wait until the task was done—thus potentially holding up the main thread while it completed—the continuation will run asynchronously. This is a better approach for composing tasks because you can write tasks that will not block the UI thread, which results in very responsive applications.

Task composability does not stop at providing continuations though, the TPL also provides considerations for scenarios, where a task must launch a number of subtasks. You have the ability to control how completion of those child tasks affects the parent task. In the following example, we will start a task, which will in turn launch a number of subtasks:

```
int[] numbers = { 1, 2, 3, 4, 5, 6 };

varmainTask = Task.Factory.StartNew(() =>
    {
        // create a new child task
foreach (intnum in numbers)
        {
int n = num;
Task.Factory.StartNew(() =>
                {
Thread.SpinWait(1000);
int multiplied = n * 2;
Console.WriteLine("Child Task #{0}, result {1}", n, multiplied);
```

```
                });
            }
        });
mainTask.Wait();
Console.WriteLine("done");
```

Each child task will write to the console, so that you can see how the child tasks behave along with the parent task. When you execute the previous program, it results in the following output:

```
Child Task #1, result 2
Child Task #2, result 4
done
Child Task #3, result 6
Child Task #6, result 12
Child Task #5, result 10
Child Task #4, result 8
```

Notice how even though you have called the `.Wait()` method on the outer task before writing **done**, the execution of the child task continues a bit longer after the task is concluded. This is because, by default, child tasks are detached, which means their execution is not tied to the task that launched it.

An unrelated, but important bit in the previous example code, is you will notice that we assigned the loop variable to an intermediary variable before using it in the task.

```
int n = num;
Task.Factory.StartNew(() =>
    {
int multiplied = n * 2;
```

If you remember our discussion on continuations in *Chapter 2, Evolution of C#*, your intuition would suggest that you should be able to use `num` directly in the lambda expression. This is actually related to the way closures work, and is a common misconception when trying to "pass in" values in a loop. Because the closure actually creates a reference to the value, rather than copying the value in, using the loop value will end up changing every time the loop iterates, and you will not get the behavior you expect.

As you can see, an easy way to mitigate this is to set the value to a local variable before passing it into the lambda expression. That way, it will not be a reference to an integer that changes before it is used.

You do however have the option to mark a child task as `Attached`, as follows:

```
Task.Factory.StartNew(
    () =>DoSomething(),
TaskCreationOptions.AttachedToParent);
```

The `TaskCreationOptions` enumeration has a number of different options. Specifically in this case, the ability to attach a task to its parent task means that the parent task will not complete until all child tasks are complete.

Other options in `TaskCreationOptions` let you give hints and instructions to the task scheduler. From the documentation, the following are the descriptions of all these options:

- `None`: This specifies that the default behavior should be used.
- `PreferFairness`: This is a hint to a `TaskScheduler` class to schedule a task in as fair a manner as possible, meaning that tasks scheduled sooner will be more likely to be run sooner, and tasks scheduled later will be more likely to be run later.
- `LongRunning`: This specifies that a task will be a long-running, coarse-grained operation. It provides a hint to the `TaskScheduler` class that oversubscription may be warranted.
- `AttachedToParent`: This specifies that a task is attached to a parent in the task hierarchy.
- `DenyChildAttach`: This specifies that an exception of the type `InvalidOperationException` will be thrown if an attempt is made to attach a child task to the created task.
- `HideScheduler`: This prevents the ambient scheduler from being seen as the current scheduler in the created task. This means that operations such as `StartNew` or `ContinueWith` that are performed in the created task, will see `Default` as the current scheduler.

The best part about these options, and the way the TPL works, is that most of them are merely hints. So you can suggest that a task you are starting is long running, or that you would prefer tasks scheduled sooner to run first, but that does not guarantee this will be the case. The framework will take the responsibility of completing the tasks in the most efficient manner, so if you prefer fairness, but a task is taking too long, it will start executing other tasks to make sure it keeps using the available resources optimally.

Error handling with tasks

Error handling in the world of tasks needs special consideration. In summary, when an exception is thrown, the CLR will unwind the stack frames looking for an appropriate try/catch handler that wants to handle the error. If the exception reaches the top of the stack, the application crashes.

With asynchronous programs, though, there is not a single linear stack of execution. So when your code launches a task, it is not immediately obvious what will happen to an exception that is thrown inside of the task. For example, look at the following code:

```
Task t = Task.Factory.StartNew(() =>
{
    throw new Exception("fail");
});
```

This exception will not bubble up as an unhandled exception, and your application will not crash if you leave it unhandled in your code. It was in fact handled, but by the task machinery. However, if you call the `.Wait()` method, the exception will bubble up to the calling thread at that point. This is shown in the following example:

```
try
{
t.Wait();
}
catch (Exception ex)
{
Console.WriteLine(ex.Message);
}
```

When you execute that, it will print out the somewhat unhelpful message **One or more errors occurred**, rather than the **fail** message that is the actual message contained in the exception. This is because unhandled exceptions that occur in tasks will be wrapped in an `AggregateException` exception, which you can handle specifically when dealing with task exceptions. Look at the following code:

```
catch (AggregateException ex)
{
foreach (var inner in ex.InnerExceptions)
    {
Console.WriteLine(inner.Message);
    }
}
```

If you think about it, this makes sense, because of the way that tasks are composable with continuations and child tasks, this is a great way to represent *all* of the errors raised by this task. If you would rather handle exceptions on a more granular level, you can also pass a special `TaskContinuationOptions` parameter as follows:

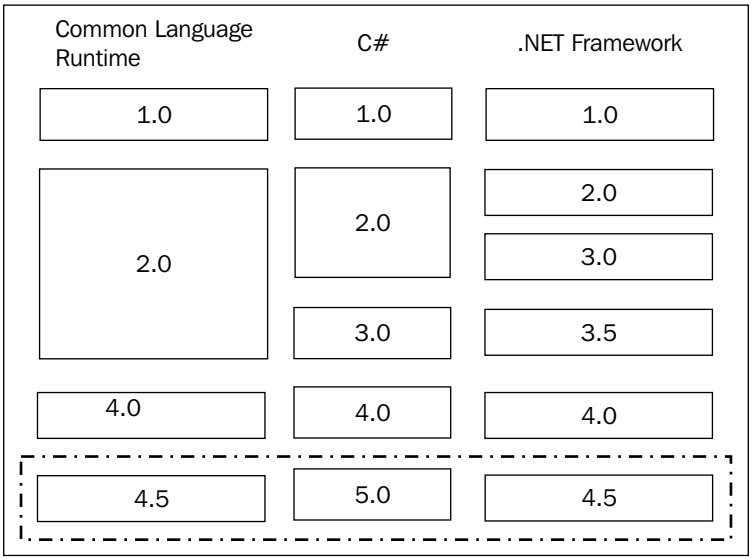
```
Task.Factory.StartNew(() =>
    {
        throw new Exception("Fail");
    }).ContinueWith(t =>
        {
            // log the exception
Console.WriteLine(t.Exception.ToString());
        }, TaskContinuationOptions.OnlyOnFaulted);
```

This continuation task will only run if the task that it was attached to is faulted (for example, if there was an unhandled exception). Error handling is, of course, something that is often overlooked when developers write code, so it is important to be familiar with the various methods of handling exceptions in an asynchronous world.

async and await

Now that the foundation for asynchrony has been set, we are ready to finally start talking about C# 5.0. The first feature we are going to discuss is quite possibly the largest impact to the way we develop applications—asynchronous programming using a new language feature that introduces the `async` and `await` keywords.

Before we go too far, let's do a quick review of the versioning situation. Although it seemed like it was going to improve when the CLR, C#, and the .NET Framework all were incremented to 4.0, it has regressed into confusing territory. The following diagram shows the comparison between the versions:



C# 5.0 comes with .NET 4.5, which also includes a new version of the Common Language Runtime. So when you develop C# 5.0 applications, you will generally be targeting the 4.5 version of the Framework.

If you have an absolute need to target Version 4.0 of the framework, you can download the *Async Targeting Pack for Visual Studio 2012*, which will give you the ability to compile and deploy your C# 5.0 applications to .NET 4.0. However, keep in mind that this is only for the C# 5.0 language features, such as async/await. The other framework updates in .NET 4.5 will not be available.

You may be asking yourself what exactly is new, considering the Task Parallel Library was introduced in the previous version of the framework. The difference is that the language itself now takes an active part in the asynchronous operation of your program. Let's start with a simple example showing the feature in action:

```
public async void DoSomethingAsync()
{
Console.WriteLine("Async: method starting");

awaitTask.Delay(1000);

Console.WriteLine("Async: method completed");
}
```

This is a very simple method from the programmer's logical perspective. It writes to the console to say **Async: method starting**, then it waits one second, and finally writes **Async: method completed**. Make special note of the two keywords in that method: `async` and `await`.

In another part of the program, we call that method writing to the console before and after we call the method as follows:

```
Console.WriteLine("Parent: Starting async method");

DoSomethingAsync();

Console.WriteLine("Parent: Finished calling async method");
```

Aside from the two new keywords, this code looks entirely sequential. Without knowing how `async` works, you might assume that the messages written to the console would come in this pattern: `parent`, `async`, `async`, `parent`. Although this is the order in which the statements are written, this is not the order in which they are executed. You can see the following example:

```
Parent: Starting async method
Child: Async method starting
Parent: Finished calling async method
Child: Async method completed
```

The statements are out of order because the method, or part of it, was executed asynchronously. What is happening here is that the compiler is analyzing the method, and literally breaking it up in such a way that everything that happens after the `await` keyword occurs asynchronously. Execution of the calling thread returns immediately and continues, and everything after the `await` call is executed in continuation.

The first reaction from most developers when they first encounter this is, "What!?"

Although it will seem hard to understand at first, once you understand how the compiler handles this, you can start to build a mental model that will help you. If we were to write that same asynchronous method using the TPL, it would look something like the following:

```
public void DoSomethingAsyncWithTasks()
{
Console.WriteLine("Child: Async method starting");

var context = TaskScheduler.FromCurrentSynchronizationContext();

Task.Delay(1000)
```

```
        .ContinueWith(t =>
            {
Console.WriteLine("Child: Async method completed");
            }, context);
}
```

In this method, we have highlighted the lines of code present in the original method. The `Task.Delay` method, which returns `Task`, is called to kick off the task (in this sample case, just waiting for one second). The next line of code is then put into a continuation, which will execute as soon as the calling task is done.

Another interesting, and perhaps more important, feature of this rewritten code is that the continuation will run on the same synchronization context as the code before the asynchronous task. So it will actually run on the same thread as the code prior to the await keyword. This becomes particularly important when you are dealing with UI code, because you cannot set property values or call UI control methods from a thread other than the main UI thread without having an exception thrown.

To be clear, this is not exactly what the compiler generates. Behind the scenes it will create a state machine that represents each stage of execution of the rewritten code. This can get very complex, when you start having loops that call and await asynchronous methods.

Despite that, the previous example is identical, logically speaking, to what the compiler generates in this case. So rather than spending a lot of time trying to explain what the compiler is doing, it is better to create a logical mental model of the behavior that you can work with.

So far you will notice that every example we have given has had the asynchronous work done in a method, and is then called by another method that awaits the value. The method, or function, is a central piece of the asynchronous puzzle. Just as you can with tasks, you can return values from asynchronous methods.

In this example, we have an asynchronous method with `Task<string>` set as the return type:

```
public asyncTask<string>GetStringAsynchronously()
{
    await Task.Delay(1000);

return "This string was delayed";
}
```

Because the method was decorated with the `async` keyword, you can return an actual string, without having to wrap it in a task. When the caller awaits the result, it will be a string, so you can treat it as a simple return type as follows:

```
public async void CallAsynchronousStringMethod ()
{
string value = await GetStringAsynchronously();

Console.WriteLine(value);
}
```

Again we see that you are able to deal with asynchronous operations, without having to worry about the infrastructure to execute them. As we showed earlier, when we rewrite the previous method to use tasks, it becomes obvious how the compiler handles the return values. Look at the following code:

```
var context = TaskScheduler.FromCurrentSynchronizationContext();

GetStringAsynchronously()
    .ContinueWith(task =>
        {
string value = task.Result;
Console.WriteLine(value);
        }, context);
```

Composing async calls

Another reason that it is helpful to think of the way the compiler rewrites async methods with tasks and continuations, is because it keeps the fact that the TPL is in use to the fore. This means that you can use the new keywords in tandem with all of the existing features of the tasks in order to parallelize your application to match your requirements. This is important to remember, because you may be missing opportunities for parallelism, if you use the `await` keyword every time.

In the following example, we are calling an asynchronous method twice. The method returns `Task<string>`, so instead of calling `await`, which would (logically) hold execution of the second task until the first one was completed, we put the return values into variables, and use the `Task.WhenAll` method to wait until they both complete as follows:

```
private async void Sample_04()
{
    Task<string>firstTask = GetAsyncString("first task");
    Task<string>secondTask = GetAsyncString("second task");
```

```
        await Task.WhenAll(firstTask, secondTask);

    Console.WriteLine("done with both tasks");
    }

    public async Task<string>GetAsyncString(string value)
    {
    Console.WriteLine("Starting task for '{0}'", value);

        await Task.Delay(1000);

        return value;
    }
```

This allows both tasks to execute at the same time, and still gives you the ability to compose your program using the await keyword.

Error handling with async methods

Error handling with asynchronous methods is very straightforward. Because the C# compiler is already rewriting the method entirely to await the completion of the task at hand before continuing, it lets you use the same exception based error handling methods that you have been using since C# 1.0.

The following is an example of an async method that throws an exception from `Task`:

```
    private async Task ThisWillThrowAnException()
    {
    Console.WriteLine("About to start an async task that throws an
    exception");

        await Task.Factory.StartNew(() =>
        {
            throw new Exception("fail");
        });
    }
```

As we discussed in the *Error handling with tasks* section, if you were interacting with the return value of this method as a regular task, then the exception would not be directly raised in the same context as the calling code. Either it will be raised when you call the `.Wait` method on the task, or you can handle it in a special continuation. But if you use `await` with the method, then you can wrap the code in a `try/catch` block as follows:

```
try
{
    await ThisWillThrowAnException();
}
catch (Exception ex)
{
Console.WriteLine(ex.ToString());
}
```

Execution of this code will seamlessly transition to the `catch` block when the unhandled exception is raised from the `async` method. This means that you don't really have to think about how you are going to handle exceptions if they are thrown from an asynchronous context, you simply `catch` them as you would if it was regular synchronous code.

Impact of async

Up to this point, we have just been discussing the mechanics of the asynchronous programming features that have been released in .Net 4.0 and C# 5.0. However, the significance of making parallel software applications easy to program deserves to be highlighted once again. There are several factors that highlight the importance of these new developments.

The First is Moore's law, which famously states that the number of transistors in CPUs is likely to double every year. While this law held true for many years, over the last decade some practical limits in cost and heat have been reached, with what is commercially possible on a single CPU. As a result, manufacturers began making computers with multiple CPUs. These new designs still manage to keep up with the prediction of Moore's law, but programs have to be specifically written to take advantage of the hardware.

Another huge factor in the impact of `async` is the rise of distributed computing. These days it is becoming more and more popular to architect programs as individual programs running on multiple computers. These peer-to-peer or client-server architectures are rarely CPU-bound, because of the latency in communicating between one computer and another over the network (or Internet). When faced with this kind of

architecture, it becomes very very important to be able to parallelize the computation so that the user interface is not left waiting for a network call to complete.

Moving forward, software applications that take advantage of opportunities to use parallelism will be the ones that are superior in performance and usability. Many of the largest Internet companies, such as Google, are already taking advantage of massive parallelization, to tackle very large problems that simply would not be computationally possible on a single computer. The `async` keyword makes it so that you almost do not have to think about how and when you take advantage of it (almost).

Improvements in .NET 4.5 Framework

In addition to all of the C# 5.0 language improvements, the .NET Framework 4.5 also brings some improvements to the table. These improvements, of course, are available to all .NET languages (that is, VB.NET), but as they become available along with C# 5.0, they warrant mention.

TPL DataFlow

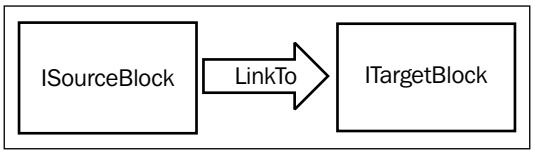
One interesting newcomer to the framework is the **TPL DataFlow** library, which aims to improve the architecture of your applications. The NuGet description for the library describes the library:

> *TPL Dataflow is a .NET Framework library for building concurrent applications. It promotes actor/agent-oriented designs through primitives for in-process message passing, dataflow, and pipelining. TDF builds upon the APIs and scheduling infrastructure provided by the Task Parallel Library (TPL), and integrates with the language support for asynchrony provided by C#.*

It can be installed via NuGet by searching for TPL DataFlow, or visiting the NuGet site at `https://nuget.org/packages/Microsoft.Tpl.Dataflow`.

As stated in the description, data flow builds on top of the Task Parallel Library, a trend that I trust you are starting to see in this release, where the TPL, and by extension `async/await` of C# 5, help you parallelize your programs; it does so without any prescription of how to structure your application at a higher level. In contrast, the TPL DataFlow library provides various building blocks for communication between disparate parts of an application.

TPL DataFlow introduces two interfaces, which like `IEnumerable` are both simple and quite deep in their implications. The following diagram shows these interfaces:



We start with the `ITargetBlock<T>`, which is a block of code that will process a number of posted messages. You will primarily interact with it by calling the `.Post` method to post a message to the block. The other side of the equation is the `ISourceBlock<T>`, which acts as a source of data. Together, these interfaces, and the concrete implementations that ship with the TPL DataFlow library, help you create applications that are structured into discrete producers, and consumers.

ActionBlock<T>

The `ActionBlock<T>` block is the simplest implementation of the `ITargetBlock<T>`. It accepts a delegate in the constructor that defines what action will be taken when a message is posted to it. The following is how you define a simple block that accepts a string and writes it to the console:

```
var block = new ActionBlock<string>(s =>
{
Console.WriteLine(s);
});
```

Once you have defined the block, you can start posting messages to it. The action block executes asynchronously, which is not a requirement, just to show how this implementation handles the posting of messages. Look at the following code:

```
for (inti = 0; i< 30; i++)
{
block.Post("Processing #" + i.ToString());
}
```

Here we see a very simple loop that iterates 30 times and posts a string to the target action. Once you have defined the target block, you can use a number of different implementations of source blocks that come with the TPL DataFlow library to create very interesting routing scenarios.

TransformBlock<T>

One such `ISourceBlock<T>` that you will find quite useful is the `TransformBlock<T, K>` block. As the name suggests, the transform block lets you take in one kind of data, and potentially transform it into another. In the following example, we will create two blocks; the `TransformBlock` will take an integer and convert it to a string. The resulting output will then be routed to `ActionBlock`, which accepts a string for processing. Look at the following example code:

```
TransformBlock<int, string> transform = new TransformBlock<int,
string>(i =>
    {
        // take the integer input, and convert to a string
        return string.Format("squared = {0}", i * i);
    });

ActionBlock<string> target = new ActionBlock<string>(value =>
    {
        // now use the string generated by the transform block
Console.WriteLine(value);
    });

transform.LinkTo(target);
```

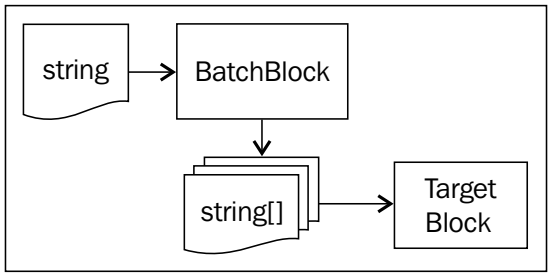
Input and output types for the transform block are designated in the form of generic parameters. You add the action block to the end of the data flow chain by using the `.LinkTo` method, which directs all the output of the source block to the target. This is explained in the following code:

```
for (inti = 0; i< 30; i++) transform.Post(i);
```

When you post an integer to the transform block, you will see that the message first flows through the transform block, and is then routed to the action block.

BatchBlock

Another kind of source block shown in the following diagram, which can help you process a stream of information, is a **batch block**:



Usually this kind of batch processing can be useful if there are certain costs associated with the processing of each message, such as informational lookups to a database. Many times in cases like this, you can batch up the query values and do a single database lookup for multiple messages at a time and amortize the cost of the lookup as you increase the batch size. Look at the following example:

```
var batch = new BatchBlock<string>(5);

var processor = new ActionBlock<string[]>(values =>
    {
Console.WriteLine("Processing {0} items:", values.Length);
foreach (var item in values)
      {
Console.WriteLine("\titem: {0}", item);
      }
    });

batch.LinkTo(processor);

for (inti = 0; i< 32; i++)
{
batch.Post(i.ToString());
}
```

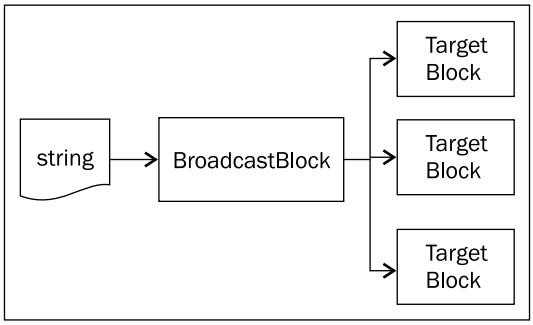
You can think of the batch block as a specific kind of transform block that takes a single instance of a message in the frontend, waits until a specified number of these messages have arrived, and then delivers that group as an array to the target block. This can be useful when you have a system that has to do some setup, such as looking up reference data for every message that it receives. If you can process many messages in one batch, then the cost of the initialization can be amortized over time. The more messages you process, the lower the cost. The following example shows how this is achieved:

```
// manually trigger
batch.TriggerBatch();
```

You can also manually trigger a batch if you know that the threshold number of messages has not been reached. In this way, you can process a batch of a smaller size if your system has to process a message within a certain amount of time.

BroadcastBlock

The broadcast block shown in the following diagram is an interesting source block:



The way it works is that you can link multiple target blocks to the broadcaster. When a message is posted to the broadcaster, it will diligently be delivered to every target. One obvious application of this block is to write a server application that has to service multiple clients at the same time. Each client is then represented by a target block that gets linked to the broadcaster. Whenever you need to notify every client, you can just post a message to the broadcaster. Look at the following example:

```
var broadcast = new BroadcastBlock<string>(value =>
{
    return value;
});

broadcast.LinkTo(new ActionBlock<string>(value =>Console.
WriteLine("receiver #1: {0}", value)));
```

```
broadcast.LinkTo(new ActionBlock<string>(value =>Console.
WriteLine("receiver #2: {0}", value)));
broadcast.LinkTo(new ActionBlock<string>(value =>Console.
WriteLine("receiver #3: {0}", value)));
broadcast.LinkTo(new ActionBlock<string>(value =>Console.
WriteLine("receiver #4: {0}", value)));

broadcast.Post("value posted");
```

In this example, we link four separate action blocks. When we post **value posted**, we will see four separate verifications of receipt in the console output. In a way, this is very similar to the existing event system in the C# language.

async I/O

To take advantage of the new `async/await` features, some very core features of the .NET Framework have evolved. Namely, the I/O features including streams, network, and file operations. This is huge because, as mentioned previously, I/O bound operations are coming to dominate the execution time of a modern application. So any improvements in the API to deal with those operations can be seen as a good sign.

At the lowest level are additions to the `Stream` API. Since .NET 1.0, this has been one of my favorite abstractions because it can be used in so many different ways. Reading and writing to a file, or a network socket, or a database, all use the stream API to represent a series of bytes of unknown size. Of course, the limiting factor here has been that, depending on the stream implementation that you are using, the performance and latency can vary greatly. So you should not write code to a network stream in the same way as code that is written to an in-memory stream, because the performance will be vastly different.

With `async` though, this changes because the `Stream` class has received new awaitable versions of all of the methods in the class. In the following example, we write an asynchronous method that takes a set of numbers, and writes them to a string as follows:

```
private static async void WriteNumbersToStream(Stream stream,
IEnumerable<int> numbers)
{
StreamWriter writer = new StreamWriter(stream);

foreach (intnum in numbers)
    {
```

```
            await writer.WriteLineAsync(num.ToString());
        }
    }
```

Although code like this would have been possible to write in a similar fashion before, the addition of methods like `.WriteLineAsync` lets you write code that is simple without having to worry about the stream holding up execution of the calling thread.

Because of the underlying improvements in the stream API, other areas, such as reading and writing files have improved. Look at the following code:

```
private static async void WriteContentstoConsoleAsync(string filename)
{
FileStream file = File.OpenRead(filename);

StreamReader reader = new StreamReader(file);
    while (!reader.EndOfStream)
    {
        string line = await reader.ReadLineAsync();
Console.WriteLine(line);
    }
}
```

I honestly cannot tell you how many times I have seen variations of this method over the years, of course, written in a non-asynchronous way. Without asynchrony, this method would absolutely choke if you attempted to read a very large file. A perfect example of this is the Notepad application that has come with every version of Windows. If you try to open a very large file, be prepared to wait because the interface will be frozen while the file is streamed from the disk.

But with the asynchronous version here, the interface will not be bogged down, regardless of the size of the file. That is the great feature of `async`, it accepts the kind of code developers are likely to write, and makes it so that common performance issues, such as buffering, will not affect the performance of the application quite as much. This is a perfect example of the "Pit of Success".

Caller attributes

One of the only non-async related improvements are **caller attributes**. In Java, there is a very common convention, which has made you specify a class level static variable called `TAG` that would contain some useful string identifier for this class as follows:

```
private static final String TAG = "NameOfThisClass";
```

Anytime you want to write information to the system log (`logcat`), you can just use the `TAG` variable so that you can easily identify the information in the log output as follows:

```
Log.e(TAG, "some log message");
```

So anytime you need to log something, the caller is responsible for self-reporting the metadata about where and why this was logged. Of course, the need to have metadata such as this for logging reaches across languages, so the C# language designers finally added a nice little feature to help you out here.

C# has always had a very powerful reflection system, so it has always been possible to take a look at the stack information in a log method. This simplifies log calls because the caller does not have to do anything special. However, this method was prone to returning unexpected results when an application was compiled in release mode, because of compiler optimizations. Also, some of the relevant classes have been excluded in portable libraries.

You can now add some compiler-optimized parameters to log methods in C# 5. When you call the method, the compiler will insert the appropriate metadata so that the correct values are returned at runtime, as follows:

```
public void Log([CallerMemberName]string name = null)
{
Console.WriteLine("The caller is named {0}", name);
}
```

The following are two other attributes that you can use:

- `[CallerFilePath]`: This gives you the path of the file in which the caller resides
- `[CallerLineNumber]`: This is the exact line number that the method was called from

Summary

In this chapter, we explored the world of asynchronous programming in C# 5 and learned the following:

- Asynchrony in software, and by extension, concurrency, is the key through which optimal performance can be unlocked.
- Task Parallel Library is the fundamental building block for all of the new asynchronous programming features. Achieving a deep understanding of the TPL will prove quite useful.
- C# 5.0 language has support for easy async, quite easily one of the most significant upgrades to the language. It builds on the TPL to make it simple to build responsive and performant applications.
- TPL DataFlow provides higher level abstractions for agent-based asynchronous programming that can help you create easy-to-maintain programs.
- Framework improvements that take advantage of the new asynchrony features, such as improvements to the I/O APIs, help you take advantage of the world of distributed computing.

Going forward, I believe that these features will make it very easy to write fast, bug-free, and maintainable programs with C#. The concepts covered here can serve as a reference for the rest of the material in this book.

4
Creating a Windows Store App

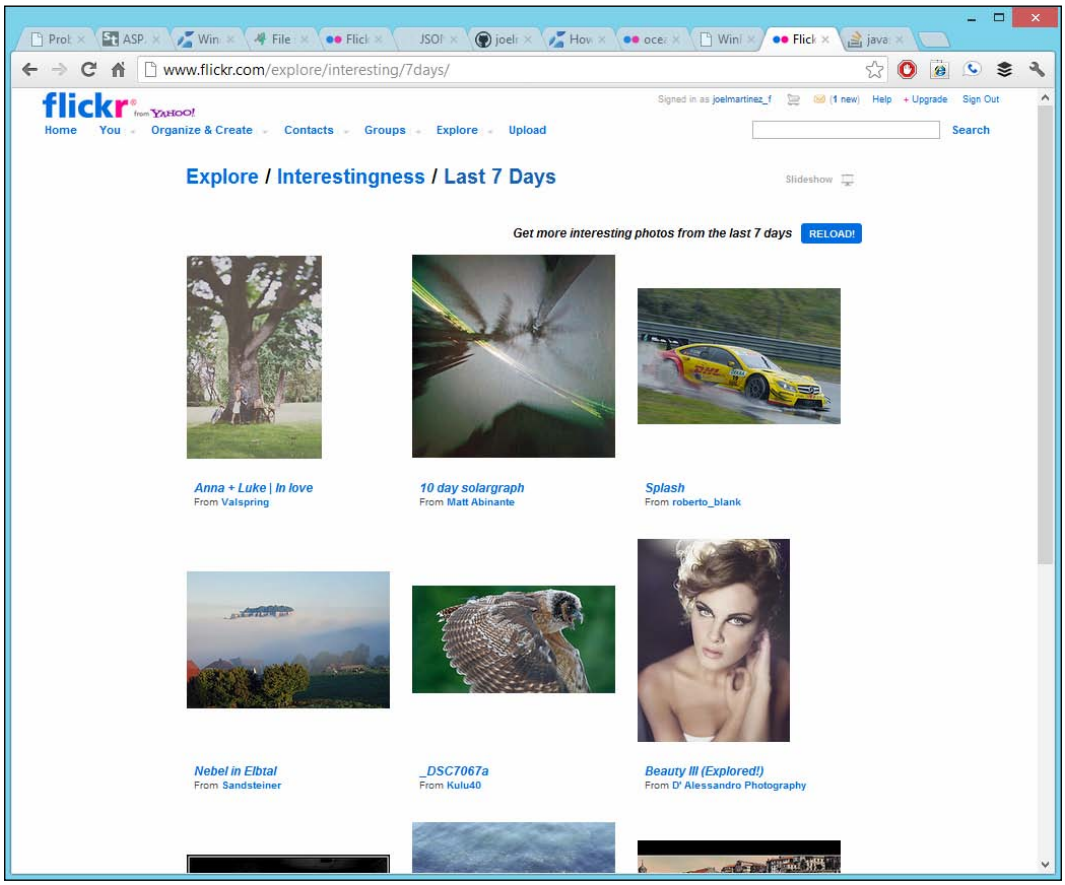
In the first half of this book, we looked at how to set up your development environment to take advantage of C# 5.0, had a look at the history and evolution of C#, and reviewed the new features available to you in the latest release. In this chapter (and for the rest of the book), we will look at some practical applications where you can use these features.

This chapter will walk you through creating a Windows Store app. This application will run in the new Windows Runtime Environment, which can target both x86 and ARM architectures. In this chapter we will create a Windows Store application, connect to an HTTP-based web service over the Internet and parsing JSON, and display the results in a XAML page.

When you are done with this chapter, you will have a fully working project that you can use as the basis for your own application. You can then upload this application to the Windows Store and potentially make money off the sales.

Making a Flickr browser

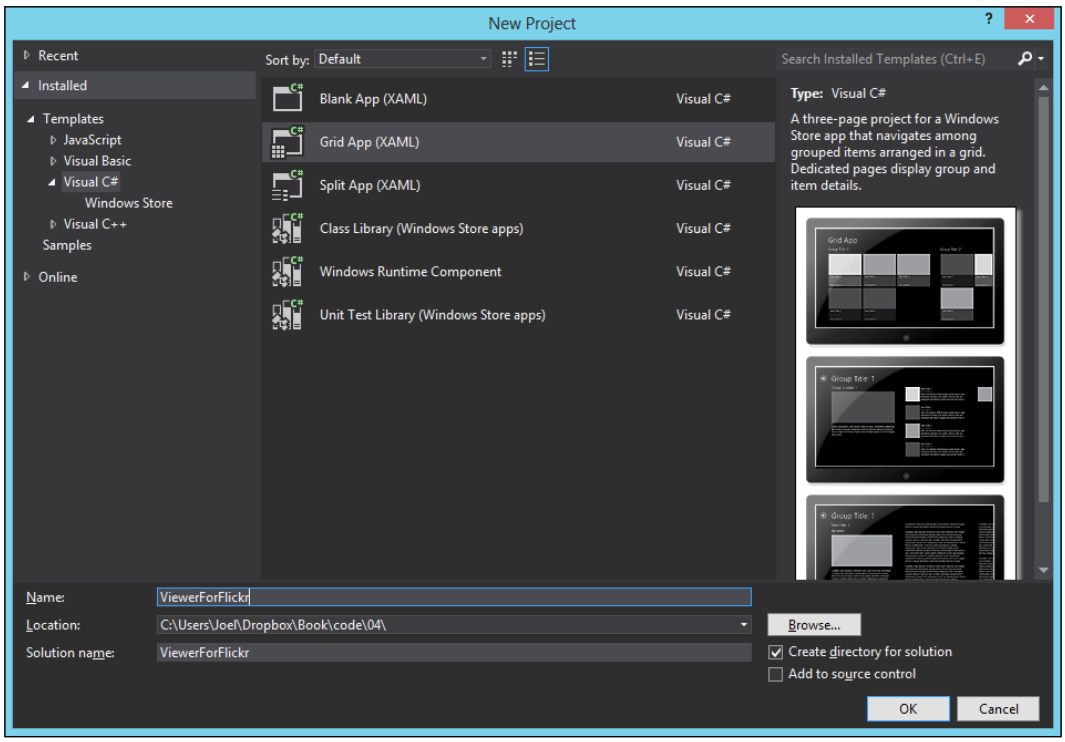
The project we are going to create is an application to browse through images. As a source, we will use the popular picture website `http://flickr.com`.



There are several reasons for choosing this as our project. First, Flickr provides an extensive web API to browse the various sections of their website, so it is an easy way to access data repositories. Secondly, many times you will find yourself accessing external data over the Internet, so this is a great example of how to handle this kind of access in a native application. And finally, pictures make for great eye candy, so the application ends up being something that you can show off.

Getting the project started

If you are using Visual Studio Express, you have to be using the version called VS Express for Windows 8. We begin the process by creating a new Windows Store application from the **New Project** dialog shown in the following screenshot:



Choose the **Blank App (XAML)** project template, which gives you the bare minimum required to create an app. You should of course feel encouraged to create projects using the other templates to see how some common UI paradigms, such as grid apps, are created. But for now, the blank app template keeps things simple.

Connecting to Flickr

Now that we have our project structure, we can begin the project by first connecting to the Flickr API. Once you have created a user account on Flickr, you can access the API documentation at `http://www.flickr.com/services/api/`.

Be sure to browse through the documentation to get an idea of what is possible. Once you are ready to continue, a key ingredient in gaining access to their data will be to provide an API key, and applying to get one is very easy—simply access the application form at `http://www.flickr.com/services/apps/create/apply/`.

Apply for a non-commercial key, and then provide information about the app that you are going to build (name, description, and so on).

Once you have completed the application, you will get two pieces of data: the API Key and API Secret. The Windows RT version of the .NET Framework contains a number of differences. For developers used to using the regular desktop version of the framework, one of those differences is the absence of the configuration system. So, although C# developers are used to entering static configuration values, such as an API Key, into an `app.config` file, we will not be able to do that here because those APIs are simply not available in WinRT applications. For a simple analog to a full configuration system, we can just create a static class to contain the constants and make them easily accessible.

We start by adding a new class to the `DataModel` namespace. If the project does not already contain it, simply add a folder named `DataModel`, and then add a new class with the following contents:

```
namespace ViewerForFlickr.DataModel
{
    public static class Constants
    {
        public static readonly string ApiKey = "<The API Key>";
        public static readonly string ApiSecret = "<The API Secret>";
    }
}
```

Of course, where it says, `<The API Key>` and `<The API Secret>`, you should replace this with the key and secret assigned to your own account.

Next up, we need a way to actually access the API over the Internet. Because of the new `async` features in C# 5.0, this is very simple. Add another class, named `WebHelper`, to the `DataModel` folder, as follows:

```
internal static class WebHelper
{
    public static async Task<T> Get<T>(string url)
    {
        HttpClient client = new HttpClient();
        Stream stream = await client.GetStreamAsync(url);

        Var  serializer = new DataContractJsonSerializer(typeof(T));
```

```
            return (T)serializer.ReadObject(stream);
        }
    }
```

Despite the small number of lines of code, there is actually a lot going on in this code. Using the `HttpClient` class, there is a single method call to download the data asynchronously. The data being returned will be in **JavaScript Object Notation** (**JSON**) format. Then, we use `DataContractJsonSerializer` to deserialize the result directly into a strongly typed class, which we define using the generic parameter on the method. That is one of the great things about C# 5.0; this method had many more lines of code in previous versions of the framework.

With the `WebHelper` class defined, we can start gathering the actual data from the remote API. One interesting API endpoint that Flickr provides is the Interesting list, which returns a list of the most interesting photos posted recently to their service. This is great because you are guaranteed to always have a fantastic set of pictures to display. You can become familiar with the method by reading through the API documentation at `http://www.flickr.com/services/api/flickr.interestingness.getList.html`.

The data that the service returns, when configured to use the JSON format, looks like the following:

```
{
    "photos": {
        "page": 1,
        "pages": 5,
        "perpage": 100,
        "total": "500",
        "photo": [
            {
                "id": "7991307958",
                "owner": "8310501@N07",
                "secret": "921afedb45",
                "server": "8295",
                "farm": 9,
                "title": "Spreading one's wings [explored]",
                "ispublic": 1,
                "isfriend": 0,
                "isfamily": 0
            }
        ]
    },
    "stat": "ok"
}
```

The object that is returned as JSON contains paging information, such as what page you are currently on, and an array of photo information. As we are going to use the built-in `DataContractJsonSerializer` class to parse the JSON result, we need to create what are called **data contracts**. These are the classes that match the structure of the object represented by the JSON string; the serializer will take the data from the JSON string and populate the properties of the data contract, so you can access it in a strongly typed fashion.

There are a number of other solutions available for working with JSON in C#. Arguably, one of the more popular solutions is James Newton-King's Json.NET, which you can find at `http://json.net`.

It is a very flexible library that can be faster than other libraries when parsing and creating JSON strings. Many open source projects take a dependency on this library. The only reason we are not using it here is for the sake of simplicity, as `DataContractJsonSerializer` comes with the framework.

We begin creating data contracts by looking at the deepest level of the JSON structure, which represents information about a single photo. A data contract is simply a class with a property for each field in the JSON object that has been decorated with some attributes. At the top of the class definition, add the `[DataContract]` attribute, which just tells the serializer that this class can be used, and then for each property add a `[DataMember(Name="<field name>")]` attribute, which helps the serializer know which members map to which JSON properties.

Compare the class in the following example code with the JSON string:

```
[DataContract]
public class ApiPhoto
{
    [DataMember(Name="id")]
    public string Id { get; set; }
    [DataMember(Name="owner")]
    public string Owner { get; set; }
    [DataMember(Name="secret")]
    public string Secret { get; set; }
    [DataMember(Name="server")]
    public string Server { get; set; }
    [DataMember(Name="farm")]
    public string Farm { get; set; }
    [DataMember(Name="title")]
    public string Title { get; set; }

    public string CreateUrl()
```

```
    {
        string formatString = "http://farm{0}.staticflickr.com/{1}/
{2}_{3}_{4}.jpg";

        string size = "m";

        return string.Format(formatString,
            this.Farm,
            this.Server,
            this.Id,
            this.Secret,
            size);
    }
}
```

The `Name` parameter passed into the data member attribute is used here because the case of the property does not match what is coming back in the JSON object. Of course, you could just name the property exactly the same as the JSON object, but then it would not match regular .NET naming conventions.

One thing that you should notice is that the photo object itself does not have a URL to the image. Flickr gives you guidance on how to construct image URLs. The `.CreateUrl` method included in the class, in the previous example, will construct the URL to the image using the information in the properties of the class. You can get more information about the different options for constructing Flickr URLs at `http://www.flickr.com/services/api/misc.urls.html`.

Next up the object chain, we have an object that contains some metadata about the result, such as the page, number of pages, and items per page. You can use this information to allow the user to page through the results. The object also contains an array of `ApiPhoto` objects, as follows:

```
[DataContract]
public class ApiPhotos
{
    [DataMember(Name="page")]
    public int Page { get; set; }
    [DataMember(Name="pages")]
    public int Pages { get; set; }
    [DataMember(Name="perpage")]
    public int PerPage { get; set; }
    [DataMember(Name="total")]
    public int Total { get; set; }
    [DataMember(Name="photo")]
    public ApiPhoto[] Photo { get; set; }
}
```

And finally, we create an object to represent the outer level object, which has just one property, as follows:

```
[DataContract]
public class ApiResult
{
    [DataMember(Name="photos")]
    public ApiPhotos Photos { get; set; }
}
```

Now that we have all of the data contracts created, we are ready to put everything together. Remember that this code will be going out over the Internet to retrieve data, which makes it a great candidate to use `async/await`. So, when we are planning out our interface, we want to make sure that it will be awaitable. Create a new class, named `Flickr.cs`, in the `Models` folder.

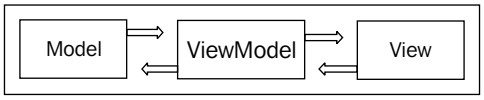
```
public static class Flickr
{
    private static async Task<ApiPhotos> LoadInteresting()
    {
        string url = "http://api.flickr.com/services/
rest/?method=flickr.interestingness.getList&api_key={0}&format=json&no
jsoncallback=1";
        url = string.Format(url, Constants.ApiKey);

        ApiResult result = await WebHelper.Get<ApiResult>(url);

        return result.Photos;
    }
}
```

In this class, we created a method called `.LoadInteresting()`, which constructs a URL pointing to the interesting endpoint using the API Key that we provisioned earlier in the chapter. Next, it uses the `WebHelper` class to make the HTTP call and passes in the `ApiResult` class, as that is the object that represents the format of the JSON result. Once the web call returns a value, it will be deserialized, and we return the photo information.

Creating the UI

We now have the data in an easy-to-access method and are ready to begin creating the user interface. When you are working with XAML, as you will be when creating apps using C# for the Windows Store (formerly known as **Metro**), a very common pattern for you to use is **Model-View-ViewModel** (**MVVM**). This architectural model is similar to **Model-View-Controller** (**MVC**), in that you have a model and

view at either end, with a component in the middle to "glue" those pieces together. It differs from MVC in that the role of the "controller" is taken by the binding system provided by XAML, which takes care of updating the view whenever data changes. So, all you have to do is provide a light wrapper around your model to make certain binding scenarios a bit easier in the form of the ViewModel. So, all you have to do is provide a light wrapper around your model to make certain binding scenarios a bit easier in the form of the ViewModel as you can see in the following diagram:



In this application, your **Model** component represents the core logic of your problem domain. The `ApiPhotos` and the `.LoadInteresting` methods represent the model in this relatively simple program. The **View** block is represented by the XAML code that we are going to create. So, we need the **ViewModel** block to link the **Model** block to the **View** block.

When you created the project, there were several bits of code that were automatically included. One of these helpful pieces of code can be found in the `Common/StandardStyles.xaml` file. This file contains a number of helpful styles and templates that you can use in your application. We are going to use one of these templates to display our images. The template named `Standard250x250ItemTemplate` is defined as follows:

```
<DataTemplate x:Key="Standard250x250ItemTemplate">
    <Grid HorizontalAlignment="Left" Width="250" Height="250">
        <Border Background="{StaticResource
ListViewItemPlaceholderBackgroundThemeBrush}">
            <Image Source="{Binding Image}" Stretch="UniformToFill"
AutomationProperties.Name="{Binding Title}"/>
        </Border>
        <StackPanel VerticalAlignment="Bottom"
Background="{StaticResource ListViewItemOverlayBackgroundThemeBrush}">
            <TextBlock Text="{Binding Title}"
Foreground="{StaticResource ListViewItemOverlayForegroundThemeBrush}"
Style="{StaticResource TitleTextStyle}" Height="60"
Margin="15,0,15,0"/>
            <TextBlock Text="{Binding
Subtitle}" Foreground="{StaticResource
ListViewItemOverlaySecondaryForegroundThemeBrush}"
Style="{StaticResource CaptionTextStyle}" TextWrapping="NoWrap"
Margin="15,0,15,10"/>
        </StackPanel>
    </Grid>
</DataTemplate>
```

Please pay attention to the way the data is being bound to the various properties of the controls of the template. This binding format automatically does a lot of the work that would normally be done in the "Controller" component of the MVC pattern. As a result, you may need to change the representation of the data in some of the models to make it easy to bind, which is why we use ViewModels.

The properties being bound in this template are different from the properties available in the `ApiPhoto` class. We will use the ViewModel to convert the model into something that can easily be bound. Go ahead and create a new class, called `FlickrImage`, that contains the properties that the template is expecting, as follows:

```
public class FlickrImage
{
    public Uri Image { get; set; }
    public string Title { get; set; }
}
```

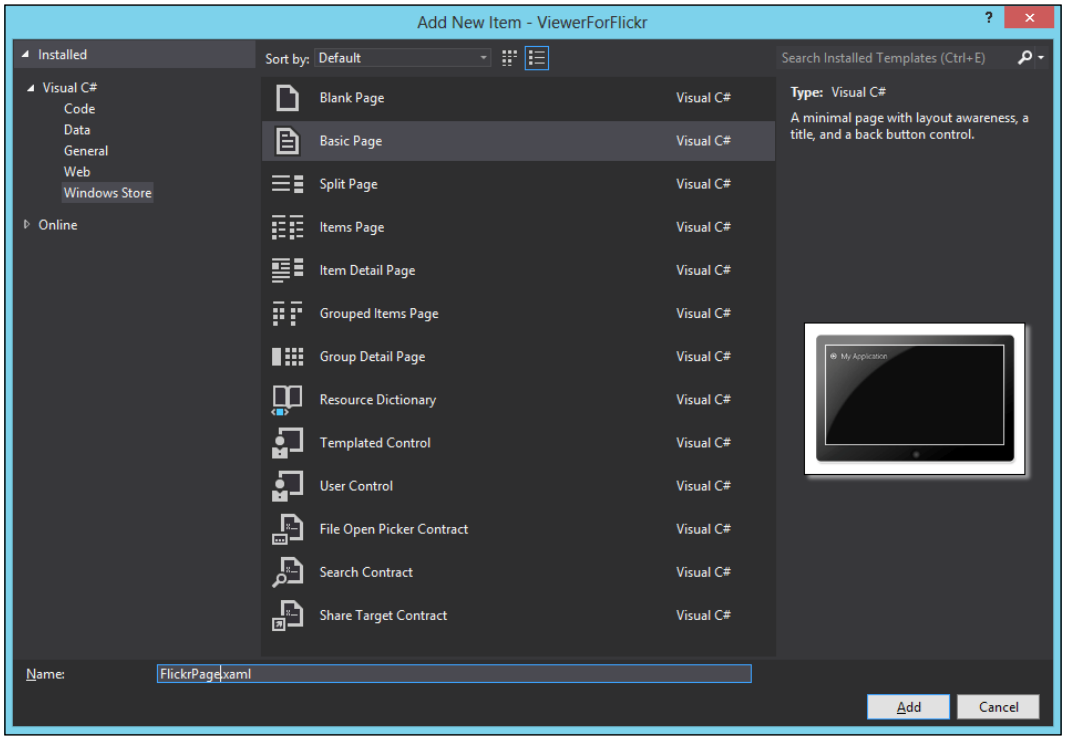
Add the following field and method to the `Flickr` class:

```
public static readonly ObservableCollection<FlickrImage> Images = new
ObservableCollection<FlickrImage>();

public static async void Load()
{
    var result = await LoadInteresting();

    var images = from photo in result.Photo
                 select new FlickrImage
                 {
                     Image = new Uri(photo.CreateUrl()),
                     Title = photo.Title
                 };

    Images.Clear();
    foreach (var image in images)
    {
        Images.Add(image);
    }
}
```

The `Load()` method starts by calling the `LoadInteresting()` method, which goes out to the Flickr API over the Internet and retrieves interesting photos (asynchronously, of course). It then converts the result into a list of the ViewModels using LINQ and updates the static `Images` property. Note how it does not reinstantiate the actual collection, instead of a regular list, the `Images` property is an `ObservableCollection`

collection, which is the preferable collection type to use in ViewModels. You can bind to the XAML UI when you initialize the page and then reload the data any time you want, the collection takes care of notifying the UI, and the binding framework takes care of updating the screen.

With the ViewModel defined, we are ready to create the actual user interface. Begin by adding a **Basic Page** item from the **Visual C# | Windows Store** menu in the **Add New Item** dialog. Name the file `FlickrPage.xaml`, as shown in the following screenshot:



This creates a very basic page with a few simple things such as a back button. You can change the page that the application starts up on by opening the `App.xaml.cs` file and changing the `OnLaunched` method to launch `FlickrPage` instead of `MainPage`, which is the default. We could, of course, have used `MainPage` for this sample, but most pages that you use will need a back button, so you should become comfortable with using the **Basic Page** template.

Our interface will be minimal, in fact, it consists of a single control that we will place in the `FlickrPage.xaml` file, below the section that contains the back button, as follows:

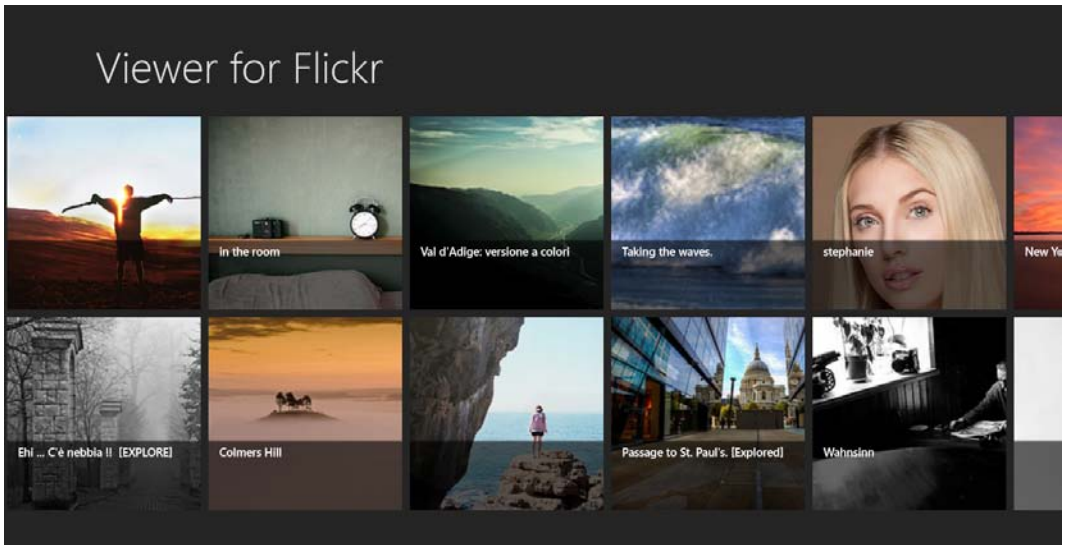
```
<GridView
    Grid.Row="2"
    ItemsSource="{Binding Images}"
    ItemTemplate="{StaticResource Standard250x250ItemTemplate}"/>
```

`GridView` will take care of displaying the items that we bind to it in a grid layout, perfect for displaying a bunch of pictures. We bind the `ItemsSource` property to our images, and `ItemTemplate`, which is used to format every single item, is bound to the standard template.

Now that the XAML has been set up, all we have to do is actually load the data and set `DataContext` for the page! Open the file named `FlickrPage.xaml.cs` and add the two following lines of code to the `LoadState` method:

```
Flickr.Load();
this.DataContext = new { Images = Flickr.Images };
```

We begin by initiating the load process. Remember that this is an asynchronous method, so it will start the process of requesting the data from the Internet. And then, we set the data context, which is what the XAML bindings use to pull their data from. See the following screenshot:



Summary

This chapter took you through the process of building a Windows Store application that uses a third party API to access information over the Internet. The asynchronous programming features of the language make it very easy to implement these features.

Here are some ideas on how you can take the application you have built here further:

- Make the application navigate to a larger version of the image when you tap on one of the thumbnails (**Hint**: Look at the `CreateUrl` method in `ApiPhoto`)
- Find out what happens when the application cannot access the Internet (**Hint**: Review the *Error handling with asynchronous methods* section in *Chapter 3*, *Asynchrony in Action*)
- Add the ability to look at different kinds of images (**Hint**: Experiment with the `Grid App` project template and look at different Flickr API methods, such as `flickr.photos.search`)
- Think about adding paging to the list of photos (**Hint**: Look at the data in `ApiPhotos` and the `ISupportIncrementalLoading` interface)

Creating native applications that run on your local machine lets you optimize for hardware that is getting more powerful with every technology cycle. Using the new asynchronous programming functionality available to you in C# 5 will ensure that you can take advantage of that hardware. In the next chapter, we look to the cloud and build a web application using the ASP.NET MVC.

5
Mobile Web App

In the previous chapter, we looked at the creation of a native desktop application, meant for distribution on the Windows Store. In this chapter, we will create a web application that lets a user log in, and see other users in the same physical area on a map. We will be using the following technologies:

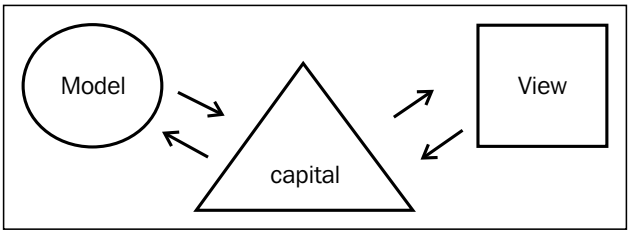
- **ASP.NET MVC 4**: This lets you build web applications using the Model-View-Controller design pattern and asynchronous programming
- **SignalR**: This is an asynchronous two-way communication framework
- **HTML5 GeoLocation**: This provides real-world location to the application
- **Client-side mapping with Google**: This is to visualize geospatial information

These technologies together let you create very powerful web applications, and with ASP.NET MVC 4—which was released along with C# 5—it is easier than ever to create mobile applications that are easily accessible to everyone over the Internet. By the end of this chapter, we will have a web application which uses modern browser features such as WebSockets lets you to connect with other web users that are physically in your vicinity. All of this makes choosing the C# technology stack a very compelling option for creating web applications.

Mobile Web with ASP.NET MVC

ASP.NET has evolved as a server platform that supports a number of different products. On the web side, we have Web Forms and MVC. On the service side we have ASMX web services, **Windows Communication Framework** (**WCF**), and Web Services, even some open source technologies, such as ServiceStack have emerged.

Development for the Web can be summarized as a melting pot of technologies. A successful web developer should be well-versed in HTML, CSS, JavaScript, and the HTTP protocol. In this sense, web development can help you develop into a polyglot programmer, someone who can work in several programming languages. We will use ASP.NET MVC for this project because of the way it applies the Model-View-Controller design pattern in the context of web development, while at the same time allowing each contributing technology the chance to do what it does best. It is shown in the following diagram:



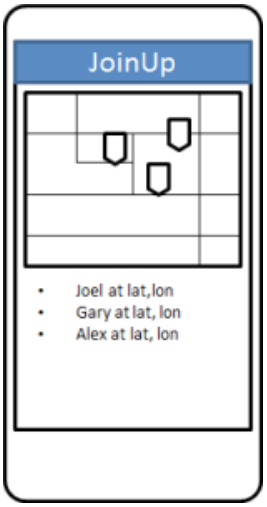
Your **Model** blocks will be all the code that contains your business logic, and the code that connects to remote services and databases. The **Controller** block will retrieve information from your **Model** layer, and pass information into it as your user interacts with the **View** block.

An interesting observation with regards to client-side development with JavaScript is that many of the application's architecture choices will be very similar to when developing any other native application. From the way you have to maintain the state of the application in memory, to the way you access and cache remote information, there are many parallels.

Building a MeatSpace tracker

On to the application we are going to build!

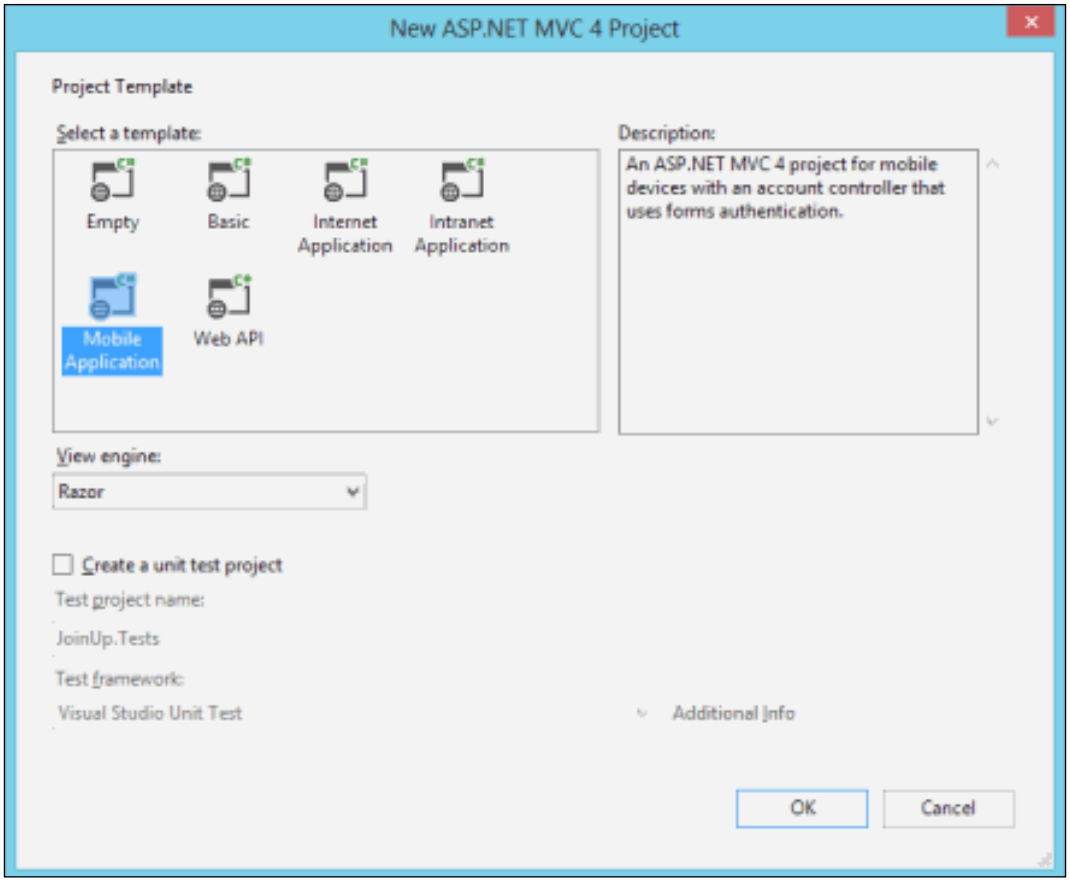
Just as the term **CyberSpace** refers to the digital realm, the term **MeatSpace** is colloquially used to refer to things or interactions that happen in the real world. The project we are going to create in this chapter is a mobile application to help you connect with other users of the web application that are physically near you. Something about the juxtaposition of building a mobile website that knows your location in the real world is very appealing, because just a few short years ago these kinds of applications were impossible on the Web.



This application will use the HTML 5 geolocation APIs to let you see other users of the application on a map. When a user connects, it will open a persistent connection to the server using SignalR, an open source project that was started by several Microsoft employees.

Iteration zero

Before we can begin writing code, we have to start the project, **iteration zero**.
We start by creating a new ASP.NET MVC 4 project, as shown in the following screenshot. In this example, I am using Visual Studio 2012 Express for Web, though of course the full version of Visual Studio 2012 will work as well.



Once you have chosen the MVC 4 project, you are presented with a dialog of several different kinds of project templates. As we want our web application to be accessible from a mobile phone, we choose one of the new project templates included in Visual Studio 2012, **Mobile Application**. This template comes preloaded with a number of helpful JavaScript libraries listed as follows:

- **jQuery** and **jQuery.UI**: This is a very popular library for simplifying access to the HTML DOM. The UI portion of the library provides a nice widget toolkit that works across browsers with controls such as date pickers.

- **jQuery.Mobile**: This provides a framework to create mobile-friendly web applications.
- **KnockoutJS**: This is a JavaScript binding framework that lets you implement the Model-View-ViewModel pattern.
- **Modernizr**: This allows you to do rich feature detection, instead of looking at the browser's user agent string to determine what features you can count on.

We will not be using all of these libraries, and of course you could use different JavaScript libraries if you so choose. But these provide a convenient starting point. You should take some time to get familiar with the files created by the project template.

The first thing you should look at is the main `HomeController` class, as this is (by default) the entry point of your application. There is some placeholder text included by default; you can easily change this to suit the application you are building. For our purposes we just change some of the text to act as simple information, and a call to action for the user to sign up.

Modify the `Views/Home/Index.cshtml` file as follows:

```
<h2>@ViewBag.Message</h2>
<p>
    Find like-minded individuals with JoinUp
</p>
```

Note the `@ViewBag.Message` header, you can change this particular value in the `Index` action method of the `HomeController` class as follows:

```
public ActionResult Index()
{
    ViewBag.Message = "MeetUp. TalkUp. JoinUp";

    return View();
}
```

There are other views which you can change to add your own information, such as the about and contact pages, but they are not critical for the purposes of this particular demonstration.

Going asynchronous

One of the most powerful additions to this latest version of ASP.NET MVC is the ability to use new `async` and `await` keywords in C# 5 to write asynchronous action methods. To be clear, you have had the ability to create asynchronous action methods since ASP.NET MVC 2, but they were rather ungainly and difficult to use.

You had to manually keep track of the number of asynchronous operations that were going on, and then let the asynchronous controller know when they were complete so that it could finalize the response. In ASP.NET MVC 4 this is no longer necessary.

As an example, we can rewrite the `Index` method that we went over in the previous section, to be asynchronous. Let's say that we wanted the message that we print in the title of the landing page to come from a database. Because that would likely be communicating with a database server on another machine, it is a perfect candidate for an asynchronous method.

First, create an awaitable method that will serve as a placeholder for retrieving the message from the database as follows:

```
private async Task<string> GetSiteMessage()
{
    await Task.Delay(1);
    return "MeetUp. TalkUp. JoinUp";
}
```

Of course in your actual code, this would connect to a database, as an example, it simply introduces a very small delay before returning the string. Now you can rewrite the `Index` method as follows:

```
public async Task<ActionResult> Index()
{
    ViewBag.Message = await GetSiteMessage();

    return View();
}
```

You can see the changes to the method highlighted in the previous code, all you have to do is add the `async` keyword to the method, make the return value a `Task<ActionResult>` class, and then use `await` in the method body somewhere. And that's it! Your method will now let the ASP.NET runtime optimize its resources as best as possible by processing other requests while it is waiting for your method to finish processing.

Getting the user's location

Once we have defined our initial landing page, we can start looking at the logged in interface. Remember that the stated goal of our application is to help you connect with other users in the real world. To do so, we will use a feature that is included in many of the modern browsers, including mobile browsers, to retrieve the user's location. To connect everyone together, we will also use a library called **SignalR**, which lets you establish a two-way communication channel with the user's browser.

The project's website describes itself simply as follows:

> *Async library for .NET to help build real-time, multi-user interactive web applications.*

With SignalR, you can write an application that lets you communicate bidirectionally to and from the user's browser. So rather than waiting for the browser to initiate communication with the server, you can actually call out and send information to the browser from the server. Interestingly, SignalR is open source, so you can dig into the implementation if you are curious. But for our purposes, we will start by adding a reference to our web application. You can do so easily through Nuget by running the following command in the package management console:

```
install-package signalr
```

Or if you would prefer to use the GUI tools, you can right-click on your project's reference nodes and choose **Manage NuGet Packages**. From there you can search for the SignalR package and click on the **Install** button.

With that dependency installed, we can begin sketching out the interface that our users will see when they are logged in, and provide us with the primary functionality of the app. We start the process of adding the new screen by adding a new controller to the `Controllers` folder using the `Empty MVC Controller` template. Name the class `MapController` as follows:

```
public class MapController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

By default, the file you create will look like the one in the previous code; make note of the controller prefix (`Map`), and action method name (`Index`). After you have created the controller, you can add the view which, as per the conventions, uses the controller name and action method name.

First, add a folder to the `Views` folder named `Map`, all views for this controller will go in here. In that folder, add a view called `Index.cshtml`. Make sure to select the `Razor` view engine, if it's not chosen already. The generated razor file is pretty bare, it just sets the title of the page (using a razor code block), and then outputs a heading with the name of the action as follows:

```
@{
    ViewBag.Title = "JoinUp Map";
```

```
}

<h2>Index</h2>
```

Now we can start modifying this view and adding the geolocation features. Add the following block of code to the bottom of `Views/map/Index.cshtml`:

```
@section scripts {
    @Scripts.Render("~/Scripts/map.js")
}
```

This scripts' section is defined in the site-wide template and makes sure to render the script reference in the correct order so that all other primary dependencies, such as jQuery, are already referenced.

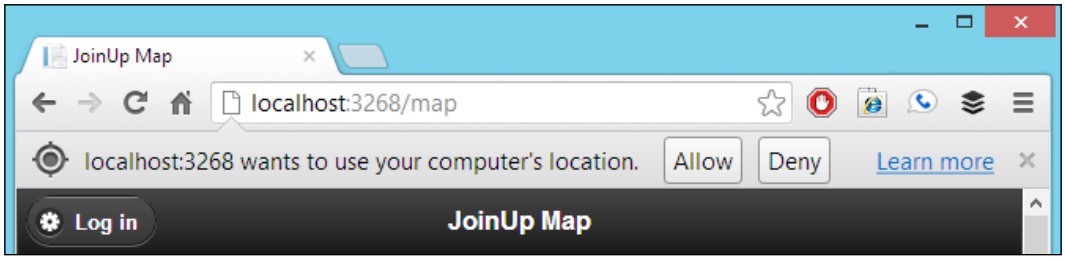
Next we create the `map.js` file that we referenced in the previous code, which will hold all of our JavaScript code. The first thing we want to do in our application is to get our geolocation working. Add the following code to `map.js` to get an understanding of how the user's location is obtained:

```
$(function () {
    var geo = navigator.geolocation;

    if (geo) {
        geo.getCurrentPosition(userAccepted, userDenied);
    } else {
        userDenied({message:'not supported'});
    }
});
```

This starts with a function definition being passed to jQuery, which will be executed when the DOM has been loaded. In that method, we get a reference to the `navigator.geolocation` property. If that object exists (for example, the browser implements geolocation), then we call the `.getCurrentPosition` method and pass in two callbacks, which we define as follows:

```
function userAccepted(pos) {
    alert("lat: " +
        pos.coords.latitude +
        ", lon: " +
        pos.coords.longitude);
}

function userDenied(msg) {
    alert(msg.message);
}
```

Once you have saved `map.js` with the previous code, you can run the web application (*F5*) to see how it behaves. As you can see in the following screenshot, the user will be prompted to accept whether they want the web application to track their whereabouts. If they click on **Allow**, the `userAccepted` method will be executed. If they click on **Deny**, the `userDenied` message will be executed. You can use this method to adjust your application accordingly, when no location is provided.



Broadcasting with SignalR

With the user's location established, the next part of the process will involve using SignalR to broadcast everybody's location to every other user that is connected.

The first thing we can do is add script references for SignalR by adding the following two lines to the script references in `Views/Map/Index.cshtml`:

```
<ul id="messages"></ul>

@section scripts {
    @Scripts.Render("~/Scripts/jquery.signalR-0.5.3.min.js")
    @Scripts.Render("~/signalr/hubs")
    @Scripts.Render("~/Scripts/map.js")
}
```

This initializes the SignalR infrastructure and allows us to build out the client side of the application before we implement the server.

At the time of writing, Version 0.5.3 of the `jQuery.signalR` library was the latest one. Depending on when you read this book, there's a good chance this version will have changed. Simply look at the `Scripts` directory after you add the SignalR dependency via Nuget to see what version you should use here.

Next, erase all of the previous contents of the `map.js` class. To keep things organized, we begin by declaring a JavaScript class with a few methods, as follows:

```
var app = {
    geoAccepted: function(pos) {
        var coord = JSON.stringify(pos.coords);
        app.server.notifyNewPosition(coord);
    },

    initializeLocation: function() {
        var geo = navigator.geolocation;

        if (geo) {
            geo.getCurrentPosition(this.geoAccepted);
        } else {
            error('not supported');
        }
    },

    onNewPosition: function(name, coord) {
        var pos = JSON.parse(coord);
        $('#messages').append('<li>' + name + ', at '+ pos.latitude
+', '+ pos.longitude +'</li>');
    }
};
```

You will recognize the `initializeLocation` method as the same code we had in there previously to initialize the geolocation API. In this version, the initialization function passes another function, `geoAccepted`, as the callback that executes when the user accepts the location prompt. The final function, `onNewPosition`, is meant to execute when somebody notifies the server of a new position. SignalR will broadcast the location and execute this function to let this script know the name of the user, and their new coordinate.

When the page loads, we want to initialize the connection to SignalR, and in the process use the object that we just created in the variable named `app`, and this can be done as follows:

```
$(function () {
    var server = $.connection.serverHub;

    server.onNewPosition = app.onNewPosition;

    app.server = server;
```

```
        $.connection.hub.start()
            .done(function () {
                app.initializeLocation();
            });
    });
```

Hubs, in SignalR, is a very simple way of exposing methods that can be invoked easily by JavaScript code on the client. Add a new class to your `Models` folder called `ServerHub` as follows:

```
public class ServerHub : Hub
{
    public void notifyNewPosition(string coord)
    {
        string name = HttpContext.Current.User.Identity.Name;

        Clients.onNewPosition(name, coord);
    }
}
```

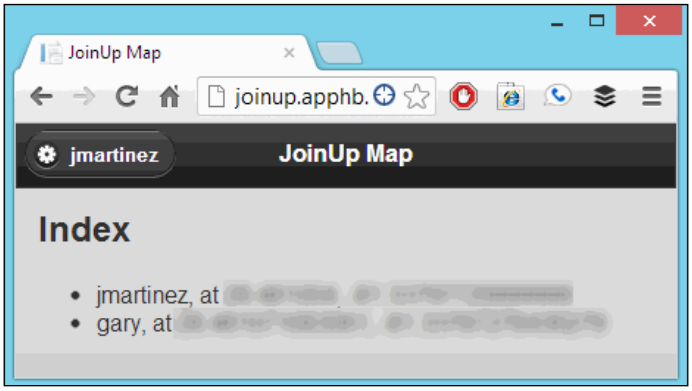
We are defining a single method in this hub, `notifyNewPosition`, which accepts a string. When we get the coordinates from a user, this method will broadcast it to all other connected users. To do so, the code first gets the user's name and then calls the `.onNewPosition` method to broadcast the name and coordinate with all connected users.

It is interesting to note that the `Clients` property is a dynamic type, so `onNewPosition` doesn't actually exist as a method on that property. The name of that method is used to automatically generate the client-side method that is called from the JavaScript code.

In order to ensure that the user is logged in when they visit the page, all we have to do is add the `[Authorize]` attribute to the top of the `MapController` class as follows:

```
[Authorize]
public class MapController : Controller
```

Press *F5* to run your application and see how we are doing. If everything is in working order, you will see a screen like the one shown in the following screenshot:



As people join the site, their location is acquired and pushed to everyone else. Meanwhile, on the client side, when a new location is received, we append a new list item element detailing the name and coordinate that was just received.

We are slowly building up our features one by one, once we have verified that this works, we can start fleshing out the next piece.

Mapping users

With location information being pushed to everyone, we can start displaying their location on a map. For this sample, we are going to use Google Maps, but you could easily use Bing, Nokia, or OpenStreet maps. But the idea is to give you a spatial reference to see who else is viewing the same web page and where they are relative to you in the world.

Start by adding an HTML element to hold the map to `Views/Map/Index.cshtml`, as follows:

```
<div
    id="map"
    style="width:100%; height: 200px;">
</div>
```

This `<div>` will serve as a container for the actual map, and will be managed by the Google Maps API. Next add the JavaScript to the scripts section above the `map.js` reference as follows:

```
@section scripts {
    @Scripts.Render("~/Scripts/jquery.signalR-0.5.3.min.js")
```

```
    @Scripts.Render("~/signalr/hubs")
    @Scripts.Render("http://maps.google.com/maps/api/
js?sensor=false");
    @Scripts.Render("~/Scripts/map.js")
}
```

As with the SignalR scripts, we just need to ensure that it is referenced before our own script (`map.js`) so that it is available in our source. Next we add code to initialize the map as follows:

```
function initMap(coord) {
    var googleCoord = new google.maps.LatLng(coord.latitude, coord.
longitude);

    if (!app.map) {
        var mapElement = document.getElementById("map");
        var map = new google.maps.Map(mapElement, {
            zoom: 15,
            center: googleCoord,
            mapTypeControl: false,
            navigationControlOptions: { style: google.maps.
NavigationControlStyle.SMALL },
            mapTypeId: google.maps.MapTypeId.ROADMAP
        });
        app.map = map;
    }
    else {
        app.map.setCenter(googleCoord);
    }
}
```

This function will be invoked when the location is obtained. It works by taking the user's initially reported location, and passing a reference to the `<div>` HTML element we created earlier with the `map` ID to a new instance of `google.maps.Map` object, setting the center of the map to the user's reported location. If the function is called again, it will simply set the center of the map to the user's coordinates.

To show all the locations, we will use a feature of Google Maps to be able to drop a marker onto the map. Add the following function to `map.js`:

```
function addMarker(name, coord) {
    var googleCoord = new google.maps.LatLng(coord.latitude, coord.
longitude);

    if (!app.markers) app.markers = {};
```

```
        if (!app.markers[name]) {
            var marker = new google.maps.Marker({
                position: googleCoord,
                map: app.map,
                title: name
            });
            app.markers[name] = marker;
        }
        else {
            app.markers[name].setPosition(googleCoord);
        }
    }
```

This method keeps track of the markers that have been added by using an associative JavaScript array, which is similar to a `Dictionary<string, object>` collection in C#. When a user reports a new position, it will take the existing marker and move it to the new position. So this means that, for every unique user that signs in, the map will display a marker and then move it every time a new location is reported.

Finally, we make three small changes to the existing functions in the app object to interact with the map. First in `initializeLocation`, we change from `getCurrentPosition` to use the `watchPosition` method as follows:

```
    initializeLocation: function() {
        var geo = navigator.geolocation;

        if (geo) {
            geo.watchPosition(this.geoAccepted);
        } else {
            error('not supported');
        }
    },
```

The `watchPosition` method will update the user's location every time it changes, which should result in a real-time view of all the locations as they report it to the server.

Next, we update the `geoAccepted` method, which is run every time the user gets a new coordinate. We can take advantage of this event to initialize the map before we notify the server of the new position as follows:

```
    geoAccepted: function (pos) {
        var coord = JSON.stringify(pos.coords);

        initMap(pos.coords);

        app.server.notifyNewPosition(coord);
    },
```

Lastly, in the method that notifies our page whenever a new position is reported by a user, we add a call to the `addMarker` function as follows:

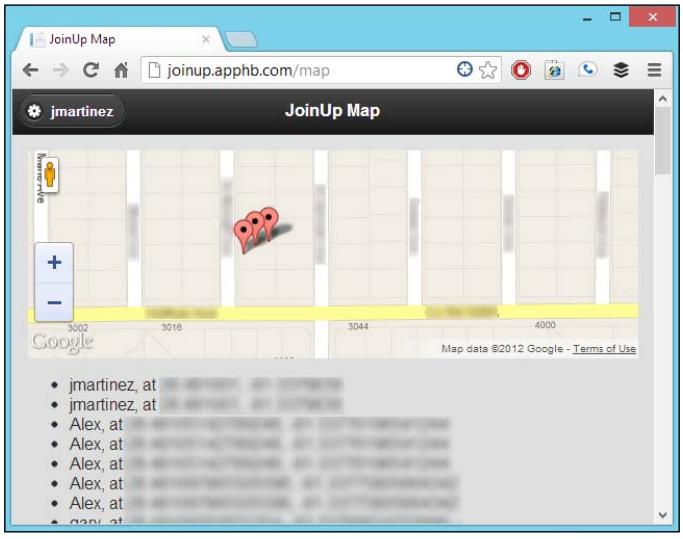
```
onNewPosition: function(name, coord) {
    var pos = JSON.parse(coord);

    addMarker(name, pos);

    $('#messages').append('<li>' + name + ', at '+ pos.latitude +', '+
pos.longitude +'</li>');
}
```

Testing the app

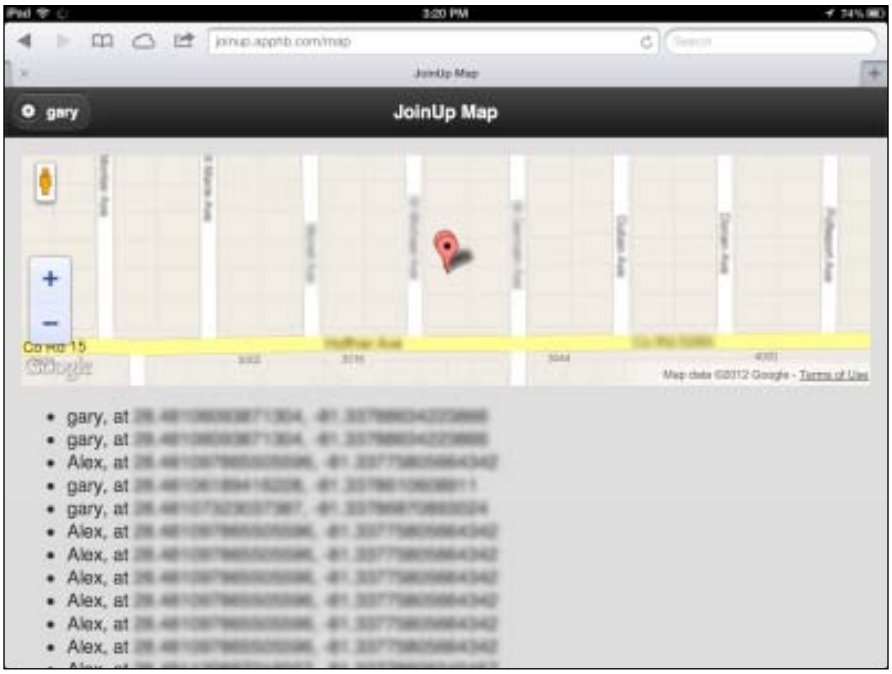
When the time comes to test the application, you can do some of the preliminary testing locally on your own computer. But it means that you will always have only a single marker in the middle of the map (that is you). In order to do a deeper test, you will need to deploy your web application to a server accessible from the Internet.

There are many options available, ranging from free (great for testing) to solutions that cost money. And of course you can always set up a server with IIS on your own and manage it that way. A great resource for finding a host can be found on the ASP.NET site at the URL `http://www.asp.net/hosting`.

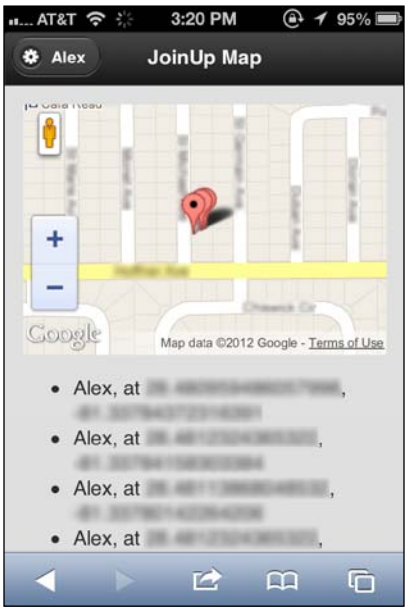
Once the application has been uploaded to the server, try accessing it from various different devices in different places. The next three screenshots demonstrate proof of the app working on the desktop:



On an iPad you will see the following screen:



And on an iPhone you will see the following screen:



Summary

There you have it… a web application that takes your physical location, and connects you in real time to other users of the app. To do this, we explored a variety of technologies that any modern web developer, and in particular, ASP.NET developer should be familiar with: ASP.NET MVC, SignalR, HTML5 GeoLocation, and client-side mapping with Google Maps.

The following are some ideas that you could use to extend this sample:

- Consider persisting the user's last known location in a database such as SQL Server or MongoDB
- Think about how you can scale this kind of application to support more users (**l**ook at the `SignalR.Scaleout` library)
- Limit the users that you notify to only those within a certain distance (learn how to calculate distance between two points on the globe with the haversine formula)
- Show points of interest that are near the user with one of the various location databases that are available on the Web such as the FourSquare Venus API, or the FaceBook Places API.

6
Cross-platform Development

Microsoft platforms are not the only platforms that can execute C# code. With the Mono framework, you can target other platforms, such as Linux, Mac OS, iOS, and Android. In this chapter, we will explore the tools and frameworks needed to build a Mac app. Some of the tools we will look at here are:

- **MonoDevelop**: This is a C# IDE that lets you write C# on other non-Windows platforms
- **MonoMac**: This provides bindings to the Mac libraries so you can use native APIs from C#
- **Cocoa**: This is the framework used to create Mac apps

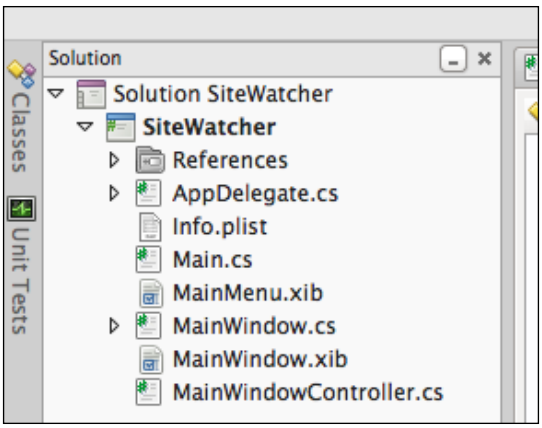
The application we are going to build in this chapter is a utility that you can use to look for text on a website. Given a URL, the application will look for links and will follow them to look for specific trigger text. We will take a look at displaying the results using Mac OS' UI SDK, AppKit.

Building a web scraper

If you have C# experience and need to build an application or utility, Mono can give you a head start at creating it quickly, using existing skill sets. Let's say you have a need to keep an eye on a website so that you can act when a new post containing a given piece of text shows up. Rather than sitting there and refreshing the page manually all day, you want to build an automated system to do this. If the website does not provide an RSS feed or other API to give you programmatic access, you can always fall back on a tried and true method of getting remote data—writing an HTTP scraper.

It sounds more complex than it is, all that this utility will do is let you put in a URL and some parameters, so the app knows what to search for. Then, it will take care of going out to the website, requesting all of the relevant pages, and searching for your target text.

Start by creating the project. Open MonoDevelop and create a new project from the **File | New | Solution** menu item which brings up the **New Solution** dialog. In that dialog, choose **MonoMac Project** from the **C# | MonoMac** list on the left-hand panel. When you create the solution, the project template initializes it with the basics of a Mac application, as shown in the following screenshot:



As with the web app we built in the previous chapter, Mac applications use the Model-View-Controller pattern to organize themselves. The project template has created the controller (`MainWindowControl`) and the view (`MainWindow.xib`); it is up to you to create the model.

Building the model

One of the primary benefits of using something like MonoMac is the ability to share code across platforms, especially if you are already familiar with C#. Because we are writing C#, any common logic and data structures can be reused if we want to build a part of the same app for a different platform. By way of example, a popular app named iCircuit (`http://icircuitapp.com`), which was written using the Mono framework, has been published for iOS, Android, Mac, and also Windows Phone. The iCircuit app achieved nearly 90 percent code reuse on some of the platforms.

The reason that this figure was not 100 percent is that one of the guiding principles that the Mono framework has been focusing on recently is building applications using native frameworks and interfaces. One of the main points of contention with cross platform toolkits in the past has been that they never feel particularly native because they are forced to settle for the lowest common denominator to maintain compatibility. With Mono, you are encouraged to use a platform's native APIs through C# so that you can take advantage of all of the strengths of that platform.

The model is where you will be able to find the most reuse, as long as you take care to keep all platform-specific dependencies out of the model where possible. To keep things organized, create a folder, named `models`, in your project, which we will use to store all of our model classes.

Accessing the Web

As with the Windows 8 application that we built in *Chapter 4*, *Creating a Windows Store App*, the first thing we want to do is provide the ability to connect to a URL and download data from a remote server. In this case, though, we just want access to the HTML text so that we can parse it and look for various attributes. Add a class, named `WebHelper`, to the `/Models` directory, as follows:

```
using System;
using System.IO;
using System.Net;
using System.Threading.Tasks;

namespace SiteWatcher
{
  internal static class WebHelper
  {
    public static async Task<string> Get(string url)
    {
      var tcs = new TaskCompletionSource<string>();
      var request = WebRequest.Create(url);

      request.BeginGetResponse(o =>
      {
        var response = request.EndGetResponse(o);
        using (var reader = new StreamReader(response.
GetResponseStream()))
        {
          var result = reader.ReadToEnd();
          tcs.SetResult(result);
        }
```

```
        }, null);

        return await tcs.Task;
      }
    }
  }
```

This is very similar to the `WebRequest` class that we built in *Chapter 4*, *Creating a Windows Store App*, except that it simply returns the HTML string that we want to parse instead of deserializing a JSON object; and because the `Get` method will be carrying out remote I/O, we use the `async` keyword. As a rule of thumb, any I/O bound method that could potentially take more than 50 milliseconds to complete should be asynchronous. 50 milliseconds is the threshold used by Microsoft when deciding which OS-level APIs will be asynchronous.

Now, we are going to build the backing storage model for the data that the user enters in the user interface. One of the things we want to be able to do for the user is save their input so that they don't have to re-enter it the next time they launch the application. Thankfully, we can take advantage of one of the built-in classes on Mac OS and the dynamic object features of C# 5 to do this in an easy way.

The `NSUserDefaults` class is a simple key/value storage API that persists the settings that you put into it across application sessions. But while programming against "property bags" can provide you with a very flexible API, it can be verbose and difficult to understand at a glance. To mitigate that, we are going to build a nice dynamic wrapper around `NSUserDefaults` so that our code at least looks strongly typed.

First, make sure that your project has a reference to the `Microsoft.CSharp.dll` assembly; if not, add it. Then, add a new class file, named `UserSettings.cs`, to your `Models` folder and inherit from the `DynamicObject` class. Take note of the `MonoMac.Foundation` namespace being used in this class, as this is where the Mono bindings to the Mac's Core Foundation APIs reside.

```
using System;
using System.Dynamic;
using MonoMac.Foundation;

namespace SiteWatcher
{
  public class UserSettings : DynamicObject
  {
    NSUserDefaults defaults = NSUserDefaults.StandardUserDefaults;

    public override bool TryGetMember(GetMemberBinder binder, out
```

```
object result)
    {
      result = defaults.ValueForKey(new NSString(binder.Name));
      if (result == null) result = string.Empty;
      return result != null;
    }

    public override bool TrySetMember(SetMemberBinder binder, object
value)
    {
      defaults.SetValueForKey(NSObject.FromObject(value), new
NSString(binder.Name));
      return true;
    }
  }
}
```

We only need to override two methods, `TryGetMember` and `TrySetMember`. In those methods, we will use the `NSUserDefaults` class, which is a native Mac API, to get and set the given value. This is a great example of how we can bridge the native platform that we are running on while still having a C# friendly API surface to program against.

Of course, the astute reader will remember that, at the beginning of this chapter, I said that we should keep platform-specific code out of the model where possible. That is, as these things usually are, more of a guideline. If we wanted to port this program to another platform, we could just replace the internal implementation of this class to something appropriate for the platform, such as using `SharedSettings` on Android, or `ApplicationDataContainer` on Windows RT.

Making a DataSource

Next, we are going to build the class that will encapsulate most of our primary business logic. When we talk about cross-platform development, this would be a primary candidate for code that would be shared across all platforms; and the better you are able to abstract your code into self-sustained classes such as these, the higher the likelihood that it will be reusable.

Create a new file, called `WebDataSource.cs`, in the `Models` folder. This class will be responsible for going out over the Web and parsing the results. Once the class has been created, add the two following members to the class:

```
private List<string> results = new List<string>();

public IEnumerable<string> Results
```

```
  {
    get { return this.results; }
  }
```

This list of strings will be what drives the user interface whenever we find a match in the website's source. In order to parse the HTML to get those results, we can take advantage of a great open source library called the **HTML Agility Pack**, which you can find on the CodePlex site (`http://htmlagilitypack.codeplex.com/`).

When you download the package and unzip it, look in the `Net45` folder for the file named `HtmlAgilityPack.dll`. This assembly will work on all CLR platforms, so you can take it and copy it right into your project. Add the assembly as a reference by right-clicking on the `References` node in the solution explorer, and choosing **Edit References | .NET Assembly**. Browse to the `HtmlAgilityPack.dll` assembly from the .NET Assembly table and click on **OK**.

Now that we have added this dependency, we can start writing the primary logic for the application. Remember, our goal is to make an interface that allows us to spider a website looking for a particular piece of text. Add the following method to the `WebDataSource` class:

```
public async Task Retrieve()
{
  dynamic settings = new UserSettings();

  var htmlString = await WebHelper.Get(settings.Url);

  HtmlDocument html = new HtmlDocument();
  html.LoadHtml(htmlString);

  foreach(var link in html.DocumentNode.SelectNodes(settings.
LinkXPath))
  {
    string linkUrl = link.Attributes["href"].Value;
    if (!linkUrl.StartsWith("http")) {
      linkUrl = settings.Url + linkUrl;
    }

    // get this URL
    string post = await WebHelper.Get (linkUrl);

    ProcessPost(settings, link, post);
  }
}
```

The `Retrieve` method, which has the `async` keyword to enable you to wait an asynchronous operation, starts by instantiating the `UserSettings` class as a dynamic object so that we can pull out the values from the UI. Next, we retrieve the initial URL and load the results into an `HtmlDocument` class, which lets us parse out all of the links that we are looking for. Here is where it gets interesting, for each link, we retrieve that URL's content asynchronously and process it.

You might assume that, because you are waiting in the loop (with the `await` keyword), each iteration of the loop will execute concurrently. But remember that asynchrony does not necessarily mean concurrency. In this case, the compiler will rewrite the code so that the main thread is not held up while waiting for the HTTP calls to complete, but the loop will not continue iterating while waiting either, so each iteration of the loop will be completed in the correct sequence.

Finally, we implement the `ProcessPost` method, which takes the contents of a single URL and searches it using the regular expression provided by the user.

```
private void ProcessPost(dynamic settings, HtmlNode link, string
postHtml)
{
  // parse the doc to get the content area: settings.ContentXPath
  HtmlDocument postDoc = new HtmlDocument();
  postDoc.LoadHtml(postHtml);
  var contentNode = postDoc.DocumentNode.SelectSingleNode(settings.
ContentXPath);
  if (contentNode == null) return;

  // apply settings.TriggerRegex
  string contentText = contentNode.InnerText;
  if (string.IsNullOrWhiteSpace(contentText)) return;

  Regex regex = new Regex(settings.TriggerRegex);
  var match = regex.Match(contentText);

  // if found, add to results
  if (match.Success)
  {
    results.Add(link.InnerText);
  }
}
```

With the `WebDataSource` class completed, we have everything we need to start working on the user interface. This goes to show how a few good abstractions (`WebHelper` and `UserSettings`) and new features such as `async` and `await` can be combined to produce relatively complex functionality, all while maintaining a great performance profile.

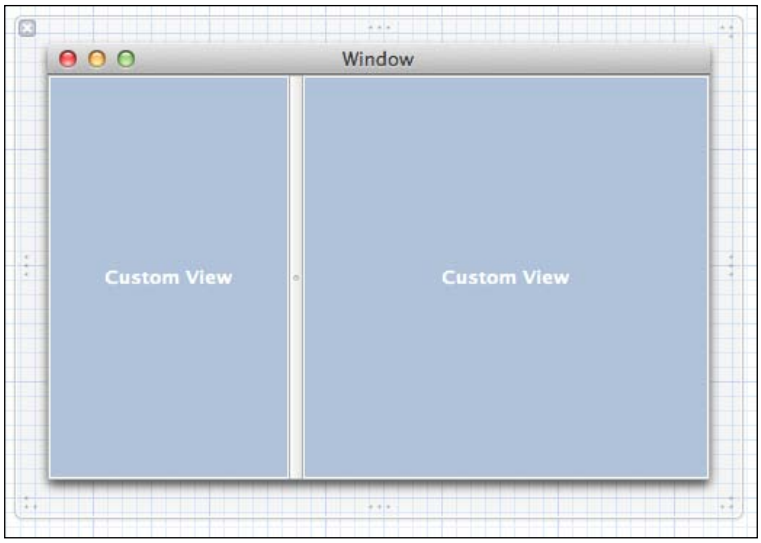
Building the view

Next, we will build the second and third legs of the MVC triangle, the view, and the controller. Starting with the view is the next logical step. When developing Mac applications, the easiest way to build the UI is to use Xcode's interface builder, which you can install from the Mac App Store. MonoDevelop on the Mac is built to specifically interoperate with Xcode for building the UI.

Start by opening `MainWindow.xib` from MonoDevelop by double-clicking on it. It will automatically open XCode with the file in the interface builder editor. The form will initially just be a blank window, but we are going to start adding views. Initially, the experience will be very familiar for anyone who has used Visual Studio's WYSIWYG editors for WinForms or XAML, but those similarities soon diverge.

If it is not already displayed, bring up the **Utilities** panel on the right-hand side of the screen by clicking on the button shown in the following screenshot, which you can find in the top-right corner of Xcode.

Find the object library and browse through the list of user interface elements that are available to you. For now, look for a Vertical Split View in the object library and drag it out to the editor surface, making sure to stretch it across the whole window, as shown in the following screenshot:

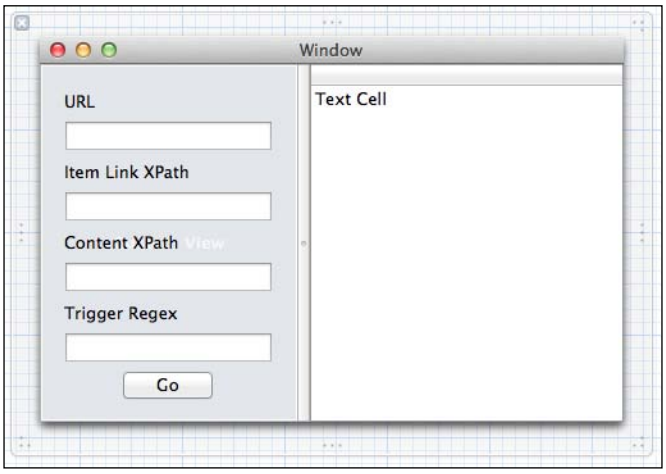


This lets us build a simple UI that lets the user resize the various elements to whatever makes sense for him/her. Next, we will add the user-provided options as text field elements, with accompanying labels, of course, to the left-hand panel.

- **URL**: This is the URL of the website that you want to scrape.
- **Item Link XPath**: This is on the page retrieved with the URL. This XPath query should return a list of links that you are interested in scanning.
- **Content XPath**: For each item, we will retrieve the HTML contents based on the URL retrieved from **Item Link XPath**. In that new HTML document, we want to pick a content element that we will look at.
- **Trigger Regex**: This is a regular expression that we will use to indicate a match.

We are also going to want a way to display the results of any matches. In order to do so, add a table view from the object library into the second right-hand panel. This table view, which is analogous to the grid controls of the regular .NET/Windows world, will give us a place to display our results in list format. Also add a push button that we will use to initiate our web call.

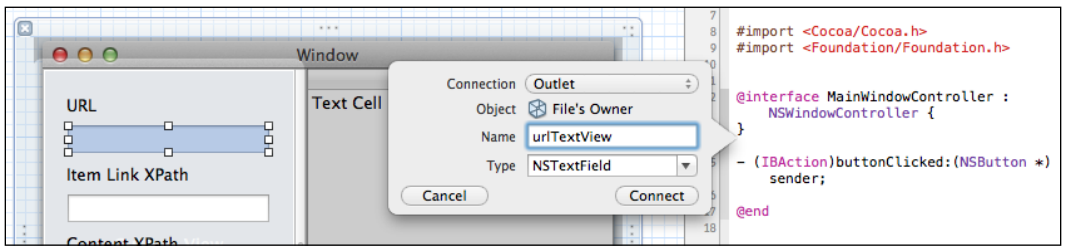
Once completed, your interface should look like the following screenshot:



With the interface defined, we start looking to the controller. Exposing the individual view elements to the controller is something unique if you have never worked with Xcode before. Where other tools for other platforms tend to automatically generate code references to textboxes and buttons, in Xcode you must manually link them up to properties in your controller. You will be exposed to some Objective-C code for this, but only very briefly, and you don't really have to do anything with it aside from the following steps:

1. Display the assistant editor and make sure that `MainWindowController.h` is showing in the editor. This is the header file for the controller that will interact with the view in our program.
2. You have to add what are called **outlets** to the controller and connect them with the UI elements, so you can get references to them from code. This is accomplished by holding the *Ctrl* key on your keyboard, and clicking-and-dragging from the control textbox into the header file.

   A small dialog, shown in the following screenshot, will be displayed, it lets you change a few options before the code is generated:



3. Do that for all of the text views and give them appropriate names such as `urlTextView`, `linkXPathTextView`, `contentXPathTextView`, `regexTextView`, and `resultsTableView`.

   When you go to add the button, you will notice that you have an option to change the connection type to an **Action** connection instead of an **Outlet** connection. This is how you can wire up the button's click event. When you have completed this, the header file should have the following elements defined:

```
@property (assign) IBOutlet NSTextField *urlTextView;
@property (assign) IBOutlet NSTextField *linkXPathTextView;
@property (assign) IBOutlet NSTextField *contentXPathTextView;
@property (assign) IBOutlet NSTextField *regexTextView;
@property (assign) IBOutlet NSTableView *resultsTableView;

- (IBAction)buttonClicked:(NSButton *)sender;
```

4. Close Xcode and go back to MonoDevelop and take a look at the `MainWindow.designer.cs` file.

   You will notice that all of the outlets and actions that you added will be represented in the C# code. MonoDevelop watches the files on the file system, and when Xcode makes changes to them, it regenerates this code accordingly.

   Remember that we want the user's settings to persist between sessions. So when the window loads, we want to initialize the textboxes with whatever values were entered previously. We will use the `UserSettings` class that we created earlier in the chapter to provide those values. Override the `WindowDidLoad` method (as shown in the following code), which is executed when the program first runs, and set the values from the user's settings to the text views.

```
public override void WindowDidLoad ()
{
  base.WindowDidLoad ();
  dynamic settings = new UserSettings();
  urlTextView.StringValue = settings.Url;
  linkXPathTextView.StringValue = settings.LinkXPath;
  contentXPathTextView.StringValue = settings.ContentXPath;
  regexTextView.StringValue = settings.TriggerRegex;
}
```

5. Now, we turn our attention to the displaying of data. Our primary output in this application is `NSTableView`, which we are going to use to display any matching links in the target URL. In order to bind data to the table, we create a custom class that inherits from `NSTableViewSource`.

```
private class TableViewSource : NSTableViewSource
{
  private string[] data;

  public TableViewSource(string[] list)
  {
    data = list;
  }

  public override int GetRowCount (NSTableView tableView)
  {
    return data.Length;
  }

  public override NSObject GetObjectValue (NSTableView tableView,
NSTableColumn tableColumn, int row)
  {
    return new NSString(data[row]);
  }
}
```

   The table view will request a row's data in the `GetObjectValue` method whenever it needs to render a given table cell. So this just takes an array of strings and returns the appropriate index from the array when requested.

6. Now we define the method that quite literally puts everything together.

```
private async void GetData()
{
  // retrieve data from UI
  dynamic settings = new UserSettings();
  settings.Url = urlTextView.StringValue;
  settings.LinkXPath = linkXPathTextView.StringValue;
  settings.ContentXPath = contentXPathTextView.StringValue;
  settings.TriggerRegex = regexTextView.StringValue;

  // initiate data retrieval
  WebDataSource datasource = new WebDataSource();
  await datasource.Retrieve();

  // display data
```

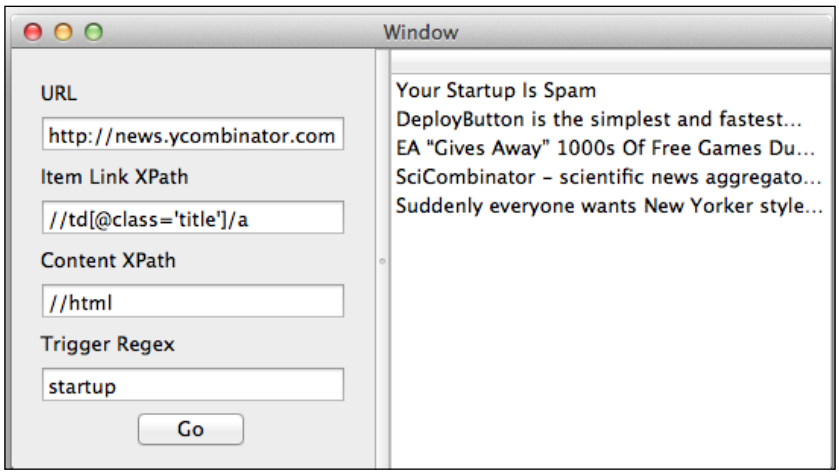
```
  TableViewSource source = new TableViewSource(datasource.Results.
ToArray());
  resultsTableView.Source = source;
}
```

In the `GetData` method, the first thing we do is pull the values from the textboxes and store them in the `UserSettings` object. Next, we retrieve the data from `WebDataSource`, asynchronously of course. Now, pass the results to `TableViewSource` so that it can be displayed.

7. Finally, implement the `buttonClicked` action that you wired up in Xcode.:

```
partial void buttonClicked (MonoMac.AppKit.NSButton sender)
{
  GetData ();
}
```

Now run the program and put in some values for a web page you want to search through. You should see results like those shown in the following screenshot, you can try to use the same values as well, but please note that it will not work if Hacker News has updated their HTML structure.



Summary

In this chapter, we created a small utility application for Mac OS, using MonoMac and MonoDevelop. Some ideas that you can use to extend or improve this application are as follows:

- Persist results across app sessions (look at Core Data)
- Build better user experience by giving the user feedback while processing (look at `NSProgressIndicator`)
- Improve the application's performance by parallelizing URL requests (look at `Parallel.ForEach`)
- Try porting the application to different platforms. For iOS, look at MonoTouch (`http://ios.xamarin.com`), and for Android, look at Mono for Android (`http://android.xamarin.com`)

C# is an incredibly expressive and powerful language. With the ability to target every mainstream computing platform, you have an incredible array of opportunities available to you as a developer, all while using a consistent programming language where you can re-use code easily across platforms.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Thank you for buying

C# 5 First Look

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

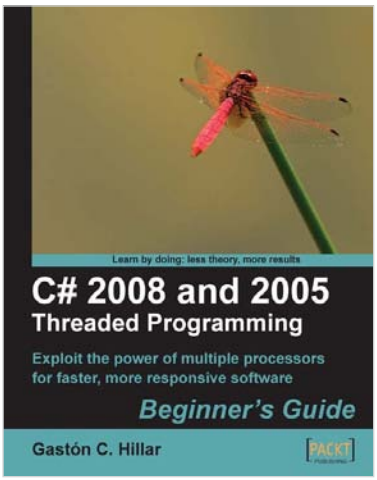
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft 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.

[PACKT] enterprise
professional expertise distilled
PUBLISHING



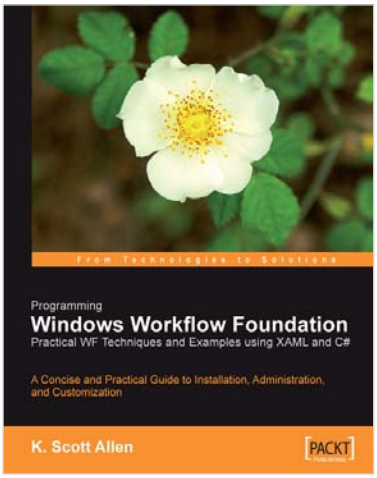
C# 2008 and 2005 Threaded Programming: Beginner's Guide

ISBN: 978-1-847197-10-8 Paperback: 416 pages

Exploit the power of multiple processors for faster, more responsive software

1. Develop applications that run several tasks simultaneously to achieve greater performance, scalability, and responsiveness in your applications
2. Build and run well-designed and scalable applications with C# parallel programming.
3. In-depth practical approach to help you become better and faster at managing different processes and threads
4. Optimized techniques on parallelized processes for advanced concepts

---



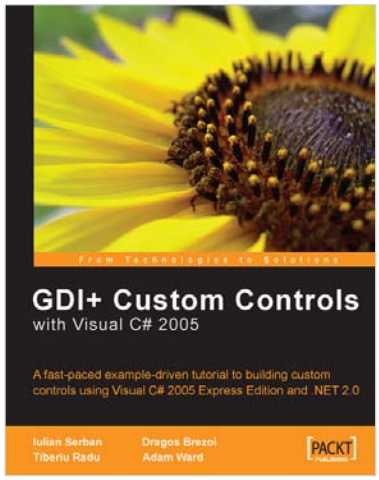
Programming Windows Workflow Foundation: Practical WF Techniques and Examples using XAML and C#

ISBN: 978-1-904811-21-3 Paperback: 252 pages

A C# developer's guide to the features and programming interfaces of Windows Workflow Foundation

1. Add event-driven workflow capabilities to your .NET applications.
2. Highlights the libraries, services and internals programmers need to know
3. Builds a practical "bug reporting" workflow solution example app

[PACKT] enterprise
professional expertise distilled
PUBLISHING

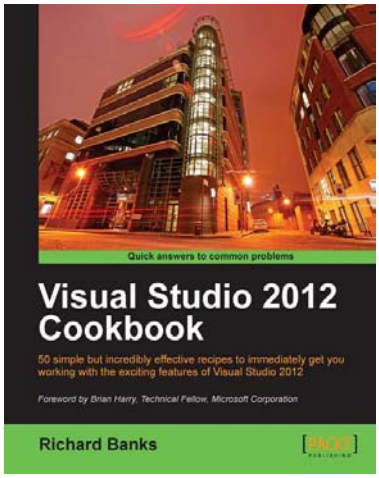


GDI+ Custom Controls with Visual C# 2005

ISBN: 978-1-904811-60-2 Paperback: 276 pages

A fast-paced example-driven tutorial to building custom controls using Visual C#2005 Express Edition and .Net 2.0

1. Learn about custom controls and the GDI+
2. Walks through great examples like PieChart control
3. Customize and develop your own controls



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