



LEARN TO CODE WITH C# AND XAML

Using Microsoft's Visual Studio 2019

Abstract

If you want to learn to code, I want to teach you. In this book, I'll teach you what you need to know so that I would hire you as a developer.

Contents

Why listen to me?	5
Why learn this, and not some other tools?	6
What is a computer?	6
What is the .NET Framework?	8
What is Windows Presentation Foundation?	9
What is C#?	9
What is XAML?	10
Some amateur psychology.....	12
The mind of a programmer.....	12
The mind of a user	12
The 'mind' of a computer	13
What is a useful program?	13
Visual Studio.....	14
Setting up your computer the way programmer would.....	16
Fn Keys	16
File Explorer	16
Item check boxes	17
File name extensions.....	17
Hidden items.....	17
Enough already, time to build a simple application	19
Creating your first project.....	19
A tour around your project	22
Displaying text in a label control.....	25
Automatically resizing the text	27
Displaying a button	28
Grid rows and columns	29
Using the editor to *finally* write some C# code.....	30
Clicking a button	30
MessageBox	30
Messages in Labels.....	32
Comments.....	33
Exercise 1	34

Adding a bit more style to our app	35
Displaying an image	35
Foreground Images	38
Showing and hiding controls.....	39
Code Cleanup	40
Exercise 2	43
Using memory for our own purposes	44
Variables.....	44
Constants	45
Variable scope.....	46
Text Input	47
Converting text to numbers.....	48
Try/Catch.....	48
Four (actually 5) operations.....	49
String concatenation	50
Formatting strings.....	52
Displaying columns of numbers/money	53
Exercise 3	53
Making decisions.....	54
If/else syntax.....	54
Boolean logic.....	54
Comparison operators	54
Compound operators.....	55
Ternary operator.....	58
Boolean Assignment	59
Switch.....	59
Enum	60
Exercise	61
More controls	62
Checkboxes	62
Radio Buttons.....	63
ListBox	64
Style.....	65

Exercise	66
Class Basics.....	68
Methods	70
Parameters.....	71
Ref	72
Out	73
Single Return value	73
Tuple returns.....	76
Using the Lambda operator	77
MVVM Basics	78
Exercise	89
Loops.....	90
While	90
For	91
Do/While	92
Populating a ListBox.....	92
Exercise	93
Writing to Files.....	95
Exercise	97
Collections.....	98
Array.....	98
List.....	100
Dictionary.....	101
HashSet	102
BindingList.....	103
ObservableCollection	105
Foreach	105
Break	105
Exercise	106
Reading and Parsing Files.....	107
Sorting.....	110
Linq.....	112
Classes in detail.....	115

Navigating Multiform Applications	123
Displaying tables of data	127
DataGrid	127
ListView	128
Accessing databases with SQL	132
Async/Await	146
Accessing databases with Entity Framework.....	155
Accessing the Internet	159

Why listen to me?

I have been paid to write software since January of 1979. I was a student on a paid internship helping the engineers at a mine calculate air flow requirements for tunnels. As I write this, that is 41 years ago. In all that time, I never stopped writing software. I was promoted to management positions, but never liked doing that work. My passion is not managing people, it is solving problems. I love puzzles and writing software is a never-ending puzzle to solve. When I started programming, the first language I was taught was Fortran. As an aspiring engineer, Fortran was a great fit since it was intended for scientific work. I also learned some assembly languages (6502, 8080, z80, among others) as the personal computer revolution was just getting started.

In 1983, I got my first full time job programming an embedded system that used an 8088 processor and the language was C. I didn't know C, but back then compilers generated two stages of output: an assembler output and a binary. I did know assembly, so by looking at the assembly that lines of C code generated, I figured out pointers and indirection.

Since then I've written drivers for Windows devices (C/C++), applications for mobile devices (C, Java, Objective-C), Web servers (JavaScript, Java, C#, PHP) and clients (TypeScript, JavaScript), and Windows desktop applications (C, C#, Visual Basic). That is the majority of the top 10 most popular languages in use today. I have been able to continue to sell my services writing software for commercial applications all this time because I continue to learn recent technologies. What I will show you in this book is a solid foundation, but you will need to keep learning as well if you want to succeed.

Why learn this, and not some other tools?

This field is still constantly changing, and today's cool new thing will be yesterday's old hat. Programmers are, to some extent, slaves to fashion and we get caught up in the new and eventually reinvent the old. I have seen enough of these cycles to know that there are somethings that are always useful to know.

As I said, I have used many technologies. My favorites are the ones I am going to lead you through in this book. They are favorites of mine because I am at my most productive when I use them. I need to offer value to those organizations that pay me to write software. I can offer the most value when I am most productive.

And I am not alone in feeling this way. Although C#, .NET, and WPF are not widely used in the startup and open source worlds, they are very widely used by Fortune 1000 companies. It does not get the buzz, but it does have a large set of developers using it every day.

Even if you eventually want to get into the startup scene, learning these tools will help you to understand the concepts of modern programming. The ideas are the same as in most other platforms and tool sets, only the expression is different. But, because of the way this toolset is constructed, you get a set of training wheels that will help you along as you start and will make you speed through the development process when your skill level improves.

So, let me introduce you to the tools we will be using.

What is a computer?

We all know what a computer is, I guess. It was something I was introduced to in college. For most of you, it is something you have used for your entire life. But you do not actually use a computer - unless you need a door stop. What you are using is the software that is loaded on the computer. When you, for example, move files, you are using the operating system (Windows, MacOS, Android). When you do your taxes, you use TurboTax. When you visit a web page, you are using Chrome or Firefox. The physical computer does nothing without software – except heat the room if you leave it on.

All modern computer hardware consists of at least one input device (keyboard, mouse, trackpad, touchscreen), at least one output device (display, printer, speakers, lights), a CPU (ARM devices in phones, Intel/AMD elsewhere), various memory (RAM for short term storage, Flash or Hard Disk for long-term storage) and some kind of network connection (Wi-Fi, Ethernet, or Cell).

An operating system is software that has embedded itself in the long-term storage device at a spot where the hardware goes to look to see what to do when power is first applied. It was not always like this. Early computers, like the digital equipment PDP 11 series, required the user to manually input the starting location in memory. This PDP-11/70 computer front panel, for example, has a series of 22 switches (numbered 0 – 21) on the left that represent a location in memory as a binary value. The switches on the right control what the computer is going to do with the information provided on the switches.



(credit: <https://wfirm.github.io/downloads/w11/pictures/HenkGooijen-pdp11-70-front.jpg>)

I remember having to do this at one of my internships. Set the switches one way, press load, set them a second way, press load again, set them a third way, press the load toggle key, and then press the run key. Off went the computer executing instructions starting at the location I had provided. When, part way through my time there, a new generation of digital machines were delivered without keys, my first thought was: How will it know where to start?

We no longer need to do that, but the principle is the same. Computers operate using a binary (1/0 or on/off) representation of the world. A bit has two states: either true or false. A single bit computer would not be especially useful but combine bits together and we have more range of expression. Eight bits in series can represent a number between 0 (all bits off) to 255 (all bits on). This is commonly referred to as a byte of data and is typically the smallest unit of memory that software deals with.

Since the English alphabet has 26 upper case letters, 26 lower case letters, and some punctuation, 256 values are enough to represent it all, plus some control characters to manipulate printer behavior. And that is what we have inherited from early machines – the ASCII character set. Visit a site like <https://www.ascii-code.com/> to see for yourself. The important thing to recognize is that these bit combinations are, for the most part, arbitrary. They are simply something that was agreed upon and for the sake of inter-computer communication. However, there is one clever bit. If you check the upper- and lower-case characters, you will see the binary pattern is identical except for the third bit from the left (bit 6, since we count from the right). For upper case characters this is a zero, for lower case characters, this is a one. This made it easy to convert upper to lower case characters and vice versa. Today, ASCII has become a subset of a worldwide character set called Unicode. The Unicode set uses 16 bits or 65536 unique values. This is managed by a standards body that includes regulating the introduction of new emojis. Visit a site like <https://unicode-table.com/> to see.

But, to return to the PDP image above. It has 22 bits of address. That is a range of 0 to 4,194,303. Notice that the section labeled Data has 16 positions. That is two bytes. So, if each unique address can reference 2 bytes of data, then this computer can work with 8 megabytes (MB) of RAM memory. A lot by the standards of the day, but the computer you have likely has at least 8 gigabytes (GB) of RAM memory or 1000 times as much.

In addition to working memory (RAM), we typically have a much larger pool of long-term storage memory. The reason we do not just have one type of memory is that hard disk memory is much slower to access than working memory. We want the computer to run as fast as possible, so we do not want it to wait for memory accesses. The operating system will transfer data stored in a binary representation (a Word document, for example) from long term storage into working memory. It will also do the same for the Word application file. Once the Word application file's data and the Word document data are both in working memory, the operating system passes control to the Word application and the CPU begins to execute instructions that end up showing you the text of the file on the screen. Then you can use your keyboard and mouse to edit the document and click on menu options to print it.

Word is an application that manipulates its data and we are going to create other applications that manipulate other data. But Word does not directly control the input and output devices connected to the computer. There are just too many different printers and screens and networks for any one program to contain all that code. Instead, modern operating systems have a concept of the 'driver'. This is a different level of software - one we will not do anything with in this book. Drivers provide a common interface so that our application code does not care whether the keyboard is hardwired, or Bluetooth, or USB. We will not care whether the long-term storage is a USB stick or a spinning hard drive or across a network.

In fact, we build our application software on top of a lot of abstractions. It just makes our life easier not to have to worry about all the common things users expect from all software so we can focus on the actual problem we are trying to solve.

What is the .NET Framework?

The first software abstraction we will encounter is a powerful one. Microsoft has created a library of standard behaviors covering all aspects of software development. They refer to this library as the .NET Framework. .NET was originally intended as a Windows only library, but its mandate has recently been expanded to work across Linux, MacOS, Android, and iOS devices as well as Windows. There are also early efforts to bring the .NET Framework to the web.

As an example, .NET provides an interface to communicate over networks. Drivers create an abstraction to describe work with a common Wi-Fi interface, for example. .NET takes it up one level and creates an abstraction that works across Bluetooth, Wi-Fi, etc.

For me, this is a major advantage of working with this tool set. The JavaScript ecosystem, for example, is dependent on open source software for its libraries. This can be great, but it is also like the wild west. There are many competing packages that have purport to have the same functionality. It is up to you to pick the 'right' one. If you later discover that the author is slow to fix bugs or add new functionality, then you need to jump elsewhere – at the cost of a major rewrite of your code. .NET, on the other hand, usually has one way of doing something and it is supported for a long time with bug fixes and new features. This stability means you spend less time evaluating other people's code and more time solving

the problem. Code I wrote 10 years ago still works unmodified – something that JavaScript projects cannot say.

What is Windows Presentation Foundation?

I have been developing Windows software for a long time. Initially, I wrote in C/C++ using the original Win32 libraries. .NET provided a wrapper around Win32 that made it easier to discover and use Win32. And .NET provides some advanced capabilities over and above Win32.

One of the main things that was made easier by .NET is working with forms. The original mechanism for this was called Windows Forms. This interface provided a way for developers to drag and drop controls visually onto a form. It had some limitations, however, in that the controls provided were difficult to customize. The controls that were provided were, in some cases, insufficient which meant creating your own custom controls.

Having learned from this, Microsoft introduced Windows Presentation Foundation (WPF) as a successor to Windows Forms. It is a modern desktop development framework that divides the application into display components and business logic components. Separating the two makes thinking and designing about each easier. It improved on the customization issues that Windows Forms suffered from.

By using a separate declarative programming language (XAML) to describe layouts – similar to HTML/CSS (but, I'd argue, more logically organized) – you get the benefits of a separate layout and style describer so your business logic is not polluted with the details of a display implementation. Most new Microsoft development technologies make use of some dialect of XAML, including web (Uno) and mobile application development (Xamarin).

It is also more performant than Windows Forms because in WPF the display component works with the Graphics Processing Unit (GPU) to render on screen components more quickly and with more functionality. Windows Forms applications do not take advantage of the GPU. Something all modern computers have available. For example, WPF can display text in any orientation (0 – 359-degree angles). To do this in Windows Forms, you are forced to write your own custom code involving bitmaps. Animations are also built in to the XAML language. Something I would never attempt in Windows Forms.

What is C#?

C# is the name of the programming language I will be using in this book. It is a compiled, strongly typed language that makes developers very productive. Up until recently, it has been used mainly for Windows Desktop and Web applications, but with the introduction of .NET Core, it is now a viable alternative for Linux, Mac, Mobile, and Web development. It is ranked in the top 5 programming languages in terms of popularity.

In the previous paragraph, I introduced two concepts that might be new to you: compiled and strongly typed.

Compiled means that the end result of using the tools to produce your application results in a file on disk (that will be loaded into working memory later) that contains the bytes of ones and zeros the CPU requires to know what to do next. The alternative to this is an interpreted language. In an interpreted language, the original programming instructions the developer wrote are provided to each computer and each computer is responsible to convert this into ones and zeros the CPU requires. There are advantages and disadvantages to both approaches.

The main advantage of the compiled approach is speed. Since the conversion to CPU understandable bytes only happens once, the program can start up and run more quickly. The downside, however, is that the application can only run on one type of CPU since ARM and Intel have completely different instruction sets (the same series of bits instruct the CPUs to do completely different things). This approach is typically used in desktop and mobile applications and is why iOS and Android app stores may not have the same applications.

The main advantage of the interpreted approach is wider accessibility. Since the conversion to CPU understandable bytes is delayed as much as possible, the same code can be made to run on diverse types of computers. Of course, we have the downside of it being slower to run. This approach is used on the web. JavaScript is sent to your browser as source code and it is up to the browser to interpret it into CPU specific bytes for the CPU it happens to be loaded onto. That is why you can open a web page on any device with a web browser and get a similar experience.

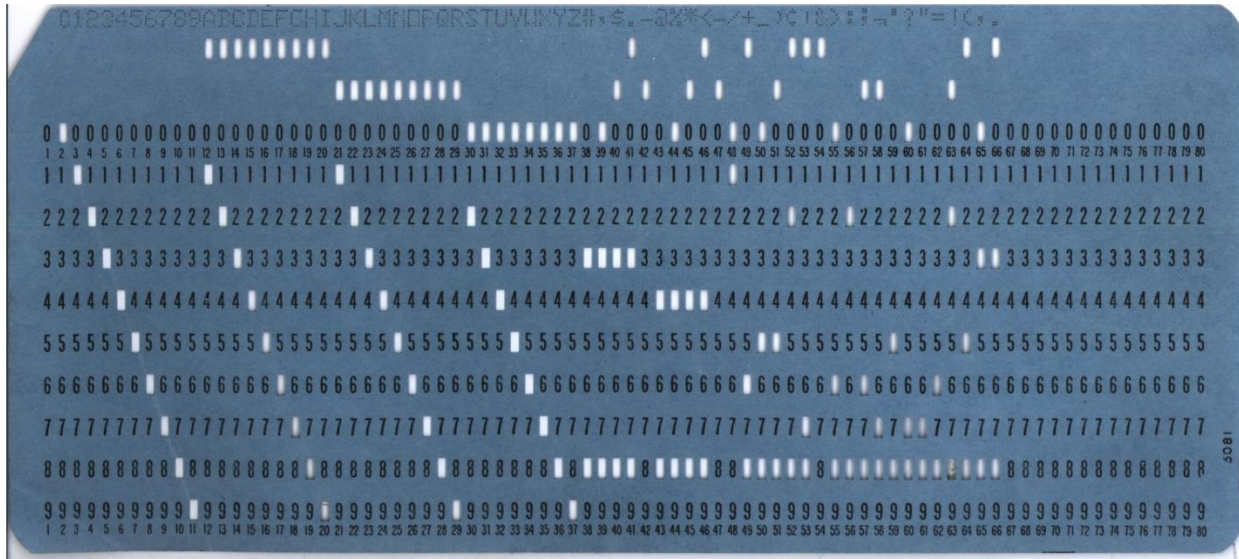
Also note that because you are shipping everyone that uses your website a copy of your source code, you will want to be careful what you put in there. Because compiled applications are already machine code, it is more difficult (but not impossible) to decipher their contents.

The other concept I mentioned – strongly typed – means that when you, through the C# language, tell the computer that you want give a name to some area of memory and have it contain some information, you need to also say what kind of information you'll be putting there. And, once you have said that, you cannot put a different kind of information there. The kinds of information are simple things like floating point numbers, integers, blocks of text (called strings), and booleans (true/false). It is also possible to build complex things from a combination of simple things. This helps you because the tools will not let you replace a number with a string, meaning that you can be sure that what is in there is always going to be a number.

JavaScript, on the other hand, is the opposite. Every named block of memory can contain any type of data at any time. Now you are responsible for ensuring that it is the right type of data when you go to use that memory in your program. This is just an extra bit of load on your brain as you think your way through your program's logic. That is why TypeScript is rising in popularity. It adds strong typing to JavaScript and makes creating large projects more manageable.

What is XAML?

Back in the days of punch cards and magnetic tape drives, data was stored in fixed sized fields. Punch cards, for example, typically had 80 columns and essentially one byte of information in each column. If you were designing a card to store someone's demographic information, you might allocate 20 columns for a first name, 20 columns for a last name, 10 for city, 20 for street address, 2 for state, 2 for country. That leaves only 6 for the birth date. (Thus, the Y2K issue was born). What if there was a city name longer than 10 characters? Oh well, too bad.



By Blue-punch-card-front.png: Gwern derivative work: agr (talk) - Blue-punch-card-front.png, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=8511203>

Once computers became more capable, people had the idea that fixed width formats should be replaced by a more flexible approach. The most popular of these formats was called eXtensible Markup Language (XML).

In XML, elements are defined by names surrounded by the <> characters. The end of an element is denoted by a name surrounded by </>. For example, <BirthDate> and </BirthDate>. All text in between the two element identifiers (tags) defines the contents of the element. XML defines some special characters – <> & – that must not appear in the text. So, these characters are represented in the text by another character sequence. For example, the character & is represented as & (I’m sure you’ve seen & in some text and wondered what was going on). So, now the text inside the BirthDate tag can be as long or as short as it needs to be to represent the data correctly.

In addition to elements, XML tags can have attributes. These appear as name/value pairs inside the opening tag. In this example, <Grid Width="7"> the Width (name) is an attribute of Grid with a value of seven.

This structured document format has been adapted for many different circumstances. For example, Hyper Text Markup Language (HTML) is based on XML. And, of course, you probably guessed by now that eXtensible Application Markup Language (XAML) is as well. We will use it to describe the layout of the user interface for our applications.

Just a word of warning, though. The source of many errors in XML and its variants is not putting the closing tag in the right place or not putting it in at all. It is good practice to create both tags first and then go back to fill in the attributes and element data.

And finally, a bit of a short cut. If your tag only has attributes and no content, then you can close the opening tag, like <Grid Width="7"/> saving you from typing some characters.

Some amateur psychology

The mind of a programmer

Good programmers are fundamentally lazy. I like to say we are ‘long term lazy’. We are always trying to find the easiest way to accomplish a task. It may cost a bit of extra time up front, but if it will minimize our effort overall, that is the path we will take. Keep that in mind and you will understand the decisions made when designing languages.

This can be at odds with the attitude of non-programmer managers, who have always tried to measure programmer productivity in not too useful ways. One of the earliest methods of deciding whether a programmer was productive was to measure their Lines of Code (LOC) output. This was always a terrible measuring stick.

I once spent a week removing 1,000 lines of code that someone else had written because they no longer did what was required of the system. Did I have negative productivity? Was the other guy extremely productive for generating that useless code?

Another time, I spent a couple of weeks reorganizing some code to make it run faster. In the end, it ran 10,000 times faster, but the lines of code stayed about the same. How can productivity be measured in this case?

The current fashion in programming circles is a process called Agile. If done well, it can be a great tool. The idea is to implement small bits of the system and put them in front of users to get feedback. If the new thing works for everyone, then great, time to move on to the next. If not, then it can be reworked right then before it becomes so encrusted with other code that it is impossible to rework without breaking a lot of other things built on top of it.

Unfortunately, that is not always the way it works. Many times, managers take this as an opportunity to micromanage developers leaving no time for anyone to look at the bigger picture. So, as you move on to careers in programming, remember that not all Agile is equal.

The mind of a user

From a programmer’s perspective, users are untrustworthy and dangerous. They are the equivalent of a three-year-old with keys and an electrical socket. If something can be made to go wrong, it will. When asked to explain what happened, they will tell you they did nothing wrong. And, even if they did acknowledge an error, it is impossible to believe what they tell you.

I once worked on a system that was deployed on mobile devices to doctors in hospitals. At the end of the day, doctors could clear their devices of patient data by pressing a button on one of the screens. Mostly this worked, but there was one doctor who claimed that patient data was being removed even though he did not press the button. We looked through the code but could not find a path that would cause this. Still the complaints continued. We sent a customer support person to shadow the doctor. She also claimed that the button had not been pressed, but the data was gone. Still, we could not figure out how this could happen. So, we added code that would log every user action and report it back to our servers and enabled this for this doctor’s device. The day after we rolled this change out, we grabbed the logs and, sure enough, it was clear that the doctor was pressing this button. And, with the logging, we could tell him exactly at what time he had done it. The complaints stopped after that. Which is to say

that humans, even those nominally predisposed to be in your favor, are notoriously unreliable witnesses and should never be trusted.

But that is not the worst part of dealing with users. When you are building a new system, you need to extract the details of what the system needs to do from users who are typically doing the function in the way they are used to. However, users will usually tell you the solution rather than the problem. It is up to you to extract the actual problem and build something to solve that.

A client once told me that the software he wanted me to modify needed to have a progress bar to let his users know how much time was left before some action they had triggered was complete so they could continue working with the software. I could have done what was asked, added a progress bar, and moved on. But I decided to have a look at what was going on. I realized that the process, though lengthy, was something that could be done in the background and would not impede the user if it were not complete. So, I suggested, and he agreed, that he forget the progress bar and move the task to the background instead. Turns out its much better to not make users wait than to let them know how long they will be waiting. The point is that the client's problem was that users were waiting for an indeterminate amount of time. His solution was to put up a progress bar. My solution was to eliminate the wait altogether. Always dig a little deeper and find the actual problem, then solve that.

The 'mind' of a computer

Computers have no mind. We might want to give them some human like qualities, but they have none. We might say, "Oh, the computer failed", but it did not. The fault lies with the programmer 99.999% of the time. The computer will do exactly what it is told to do, without question. You, the programmer, are in total control so any failures are on you.

Twenty years ago, you might have had a case that the tools had bugs and were letting you down. Today, the tools are pretty much bullet proof. You are more likely to be struck by lightning than encounter a bug in the compiled output of the tool.

What is a useful program?

Useful programs take input and process it in some way to produce an output. Input can be in the form of mouse and keyboard, file, network message received, or something like the current time. Output can be in the form of a display, a file, a network message sent, or an audio signal.

If you are not transforming the data in some way, then you are not doing something useful. All code you write should have that goal. If it does not, then be ruthless and get rid of it.

Visual Studio

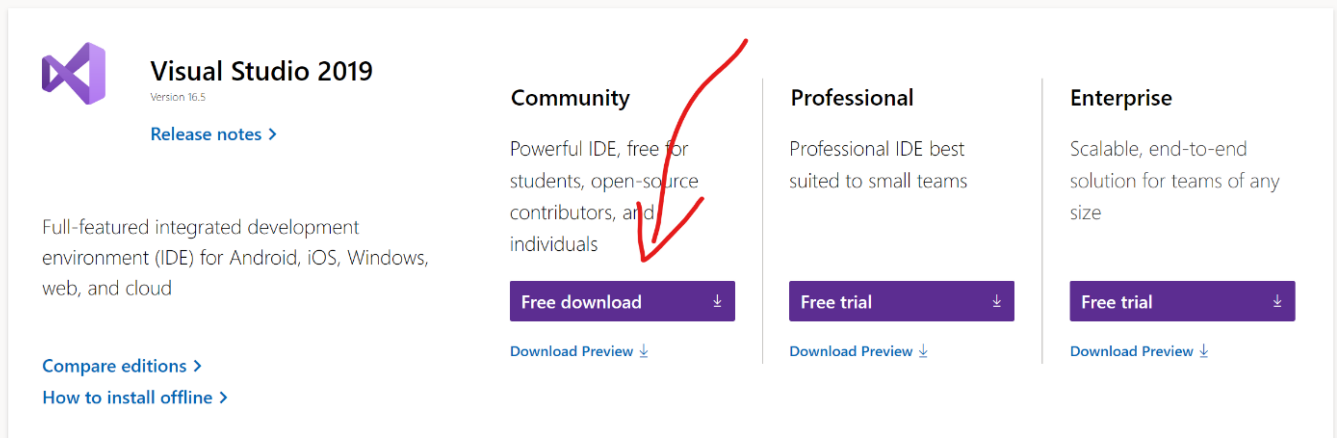
Visual Studio is an advanced development tool that takes care of much of the administrative tasks associated with the development process. This is the flagship Microsoft developer product and any kind of application Microsoft tools allow you to build can be built here. In my mind, it is the best development tool I have ever used and whenever I cannot use it, I wish I could.

Please be aware that Visual Studio only works in Windows. If you are on a Mac or Linux system, you will need to dual boot into Windows to use it. Visual Studio for Mac is not the same product. Setting up dual boot is beyond the scope of this book, but, for Mac, you should have a look at Boot Camp or Parallels.

With that out of the way, go to <https://visualstudio.microsoft.com/downloads/> and download the Community Edition. I am suggesting the Community Edition because it is free. It also has all the features that you will need as a solo developer. The Professional and Enterprise editions provide additional capabilities for large groups of developers, but you will not need any of those to work through this book. If you do, the organization you work for will likely have a site license for all their programmers and you will get your copy that way.

As of this writing, the latest version is Visual Studio 2019 and the download page appears like this:

Downloads



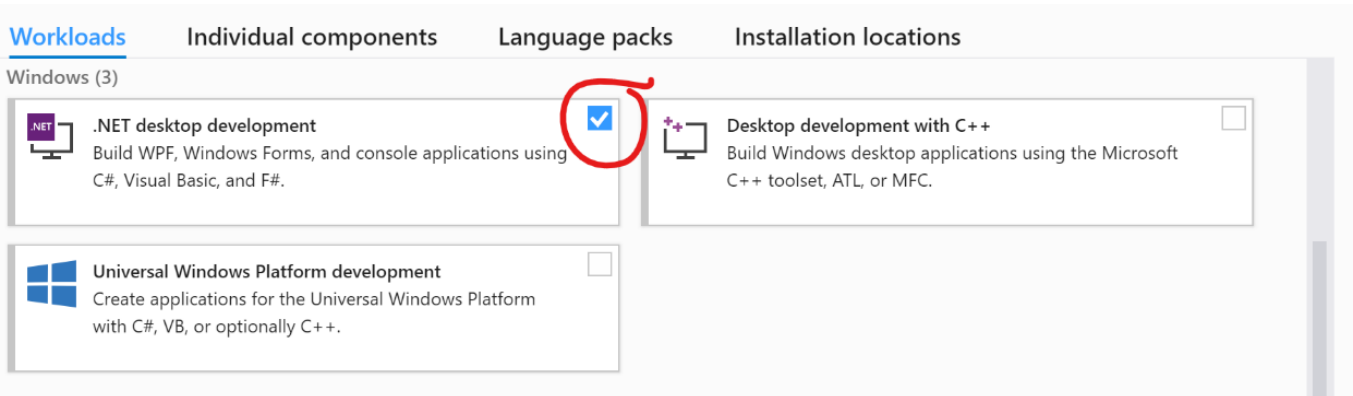
Visual Studio 2019
Version 16.5
[Release notes >](#)

Full-featured integrated development environment (IDE) for Android, iOS, Windows, web, and cloud

[Compare editions >](#)
[How to install offline >](#)

Community	Professional	Enterprise
Powerful IDE, free for students, open-source contributors, and individuals	Professional IDE best suited to small teams	Scalable, end-to-end solution for teams of any size
Free download ↓	Free trial ↓	Free trial ↓
Download Preview ↓	Download Preview ↓	Download Preview ↓

Once downloaded, it will begin to install. At one point in the installation process, you should see the Visual Studio Installer displaying a screen like this one asking you to pick the workloads to install.



Ensure that at least .NET Desktop Development is checked to follow along with the examples in this book. Installing other workloads will allow you to create web applications and work with Azure cloud services, among other things. It will not harm anything to install more workloads, but it will take up more space on your computer.

Setting up your computer the way programmer would

Fn Keys

As you get more familiar with Visual Studio, you will want to use keyboard shortcuts for common operations. Many of the shortcuts use the F keys – that top row of keys on your keyboard that you currently use to adjust the audio volume and screen brightness. You can use your computer's Fn key to get at the F keys (press Fn+F1, for example, and you will get a help dialog in most applications).

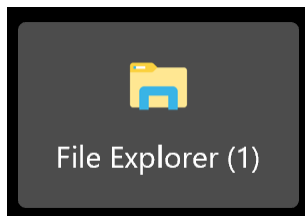
However, as a programmer, you will be needing access to shortcuts much more frequently than the number of times you adjust your computer's volume or screen brightness. So, I recommend you alter the default behavior of those keys so that Fn+Fx accesses the multimedia controls and Fx accesses the function keys.

I cannot explain how to do this for every possible Windows machine out there, but the author of this link: <https://www.howtogeek.com/235351/how-to-choose-whether-your-function-keys-are-f1-f12-keys-or-special-keys/> does a pretty good job of directing you on how to do it for your computer. It isn't necessary, but as you get more familiar with Visual Studio and start wanting to go faster, moving your hands off the keyboard to work the mouse or pressing that extra Fn key will start to feel like its slowing you down.

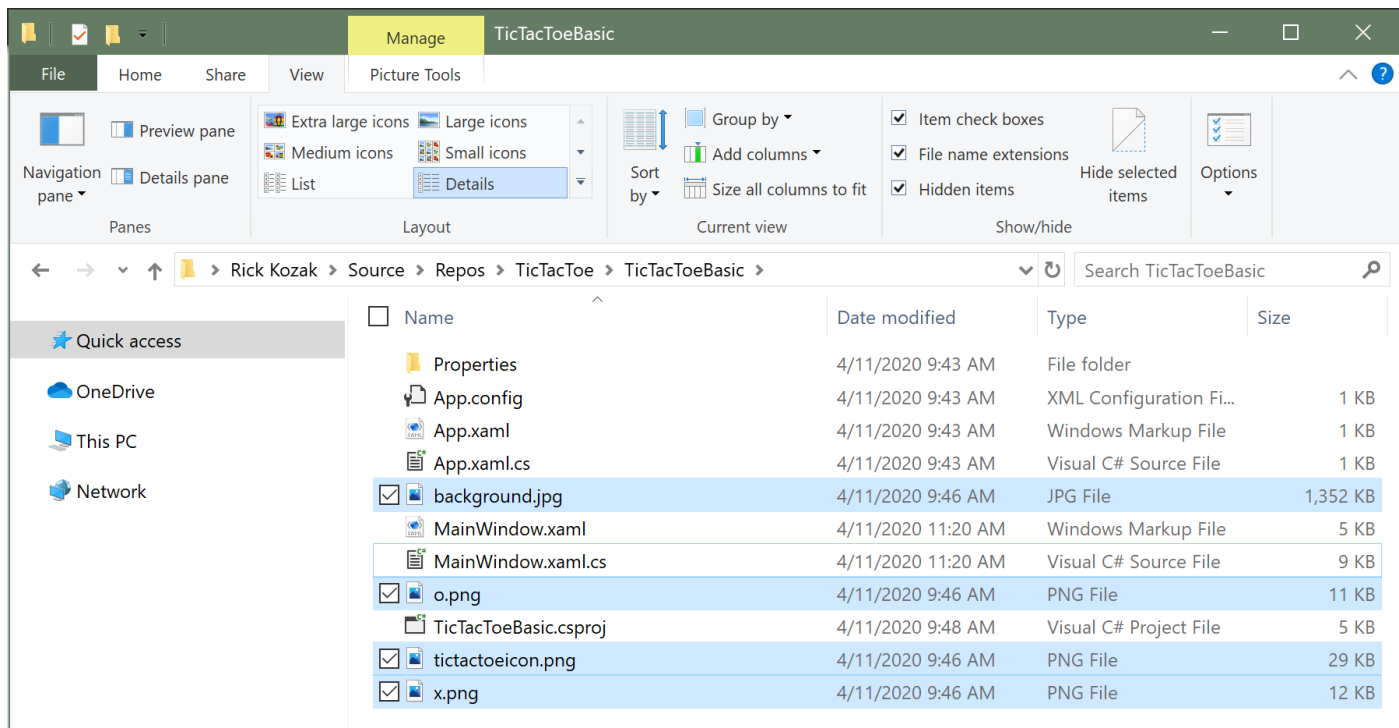
This is also my first example of what I mean by 'long term lazy'. A little bit of extra work now will, over time, save you lots of potential Fn key presses.

File Explorer

There are three settings in Windows File Explorer that I recommend you change. To get at them, start File Explorer by clicking on the icon in the Start popup or however you usually do it.



And then click the View tab and look at the Show/hid area. There you will see three options: Item check boxes, File name extensions, and Hidden items. You will want to check all three, as shown here:



Item check boxes

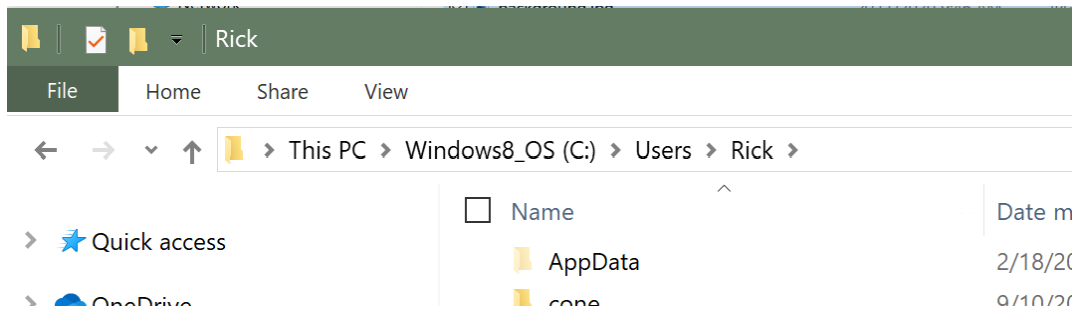
Although you can select many non-consecutive files at a time by holding down the ctrl key and clicking on the file, just one click without the ctrl key pressed and you lose all your work. So, I recommend turning on the 'Item check boxes' option to allow you to select multiple arbitrary files with just a click.

File name extensions

As a developer, there will be many times where you will want to create a file with a specific extension or adjust an existing file's extension. By default, Windows does not show you the file extension. What happens is that a file with the name 'x.txt' appears in File Explorer as 'x'. If you rename that file, to say 'x.csv' thinking that you will now have a new file extension, you will be wrong. That is because Windows renamed the file to 'x.csv.txt' – that is, it kept the hidden extension hidden. Change this option so you can see and modify the entire file name, including extensions.

Hidden items

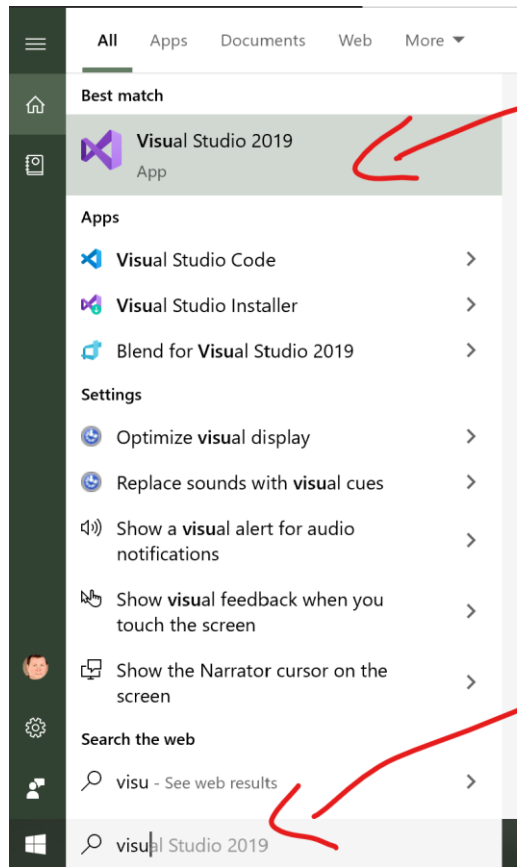
As a developer, you want to have access to files and directories on your computer that are hidden from normal users. Turn on this option to have File Explorer show you those usually hidden items. You will be able to tell that they are normally hidden, as they will appear washed out compared to normally visible items. Once you've turned this on, use File Explorer to go to C:\Users\<your username>.



In my case, Rick is my user's name. Notice the AppData directory with its slightly faded folder icon. That is a hidden directory you can now access. If you go in there and then one more level down into Roaming, you will find a directory for every application you have installed on your computer. This is where applications store private data they want kept hidden from users. Eventually, we will learn how to put our own data here.

Enough already, time to build a simple application

Creating your first project



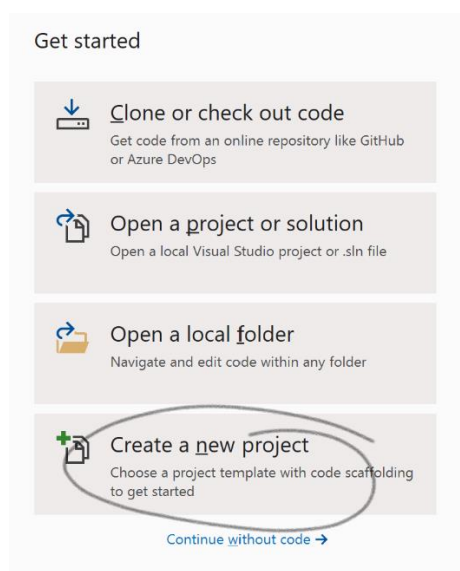
Now that Visual Studio 2019 is installed on your computer, you should be able to find it and start it up. From the search bar in Windows, start typing 'visu....' At some point along the way, it should be offering you a 'Best Match' – and that is what you are looking for. The purple stylized mobius strip icon is what you want to click on to get Visual Studio running.

If you right click on this, the context menu that pops up allows you to 'Pin to taskbar' and 'Pin to start'. My preference is to pin the icon to the taskbar. Then it is always available along the bottom of the screen (or wherever you place your taskbar) for single click access.

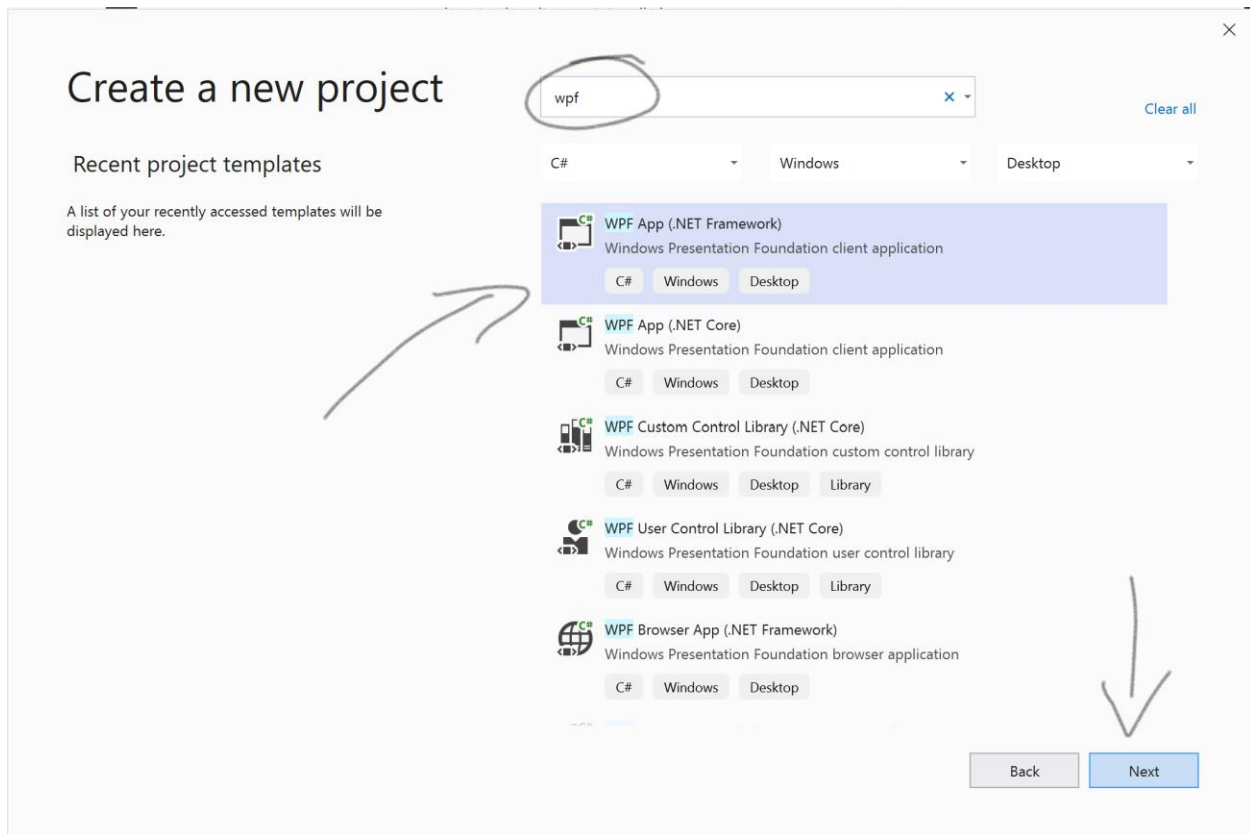
Also notice that Visual Studio Installer is listed as an alternative app. Use this if you want to add or remove workloads in the future. It is also where you go to update the Visual Studio application.

Microsoft is making constant improvements and there always seems to be an updated version.

Once started, there will be a slight delay as it does some one-time setup. Once that is complete, you should see a dialog that looks like this. Select the 'Create a new project' option.



That leads you to the next screen that looks like this.



What I have done in the example is entered 'wpf' into the search box to filter the list of project types. Even adding the 'C#', 'Windows', and 'Desktop' filters still results in a few possible project types. We will not be creating a library – those are blocks of common code to be incorporated into multiple projects – so we can drop those. The WPF Browser App is an older technology that attempts to run a .NET Framework app in Chrome, Firefox, or Edge. None of these browsers currently support this. Although it is still supported in IE11, it is best to ignore this project type.

The remaining two options WPF App (.NET Framework) and WPF App (.NET Core) are effectively identical. The current version of .NET Framework is 4.x. The current version of .NET Core is 3.x. Shortly, there will be a .NET 5 that will merge these two libraries, but for now they are parallel paths to the same destination. I have used .NET Framework for all the examples in this book, so I suggest you pick that option and click next. However, if you do select the Core option, you should not have any problem following along.

Note that once you've picked an option, it will show up on the left in 'Recent project templates' and you can just pick it from there without searching.

Now you are faced with this dialog where you need to provide a name to this project in the 'Project Name' textbox.

×

Configure your new project

WPF App (.NET Framework) C# Windows Desktop

Project name

Location

 ...

Solution name ⓘ

☐ Place solution and project in the same directory

Framework

Back Create

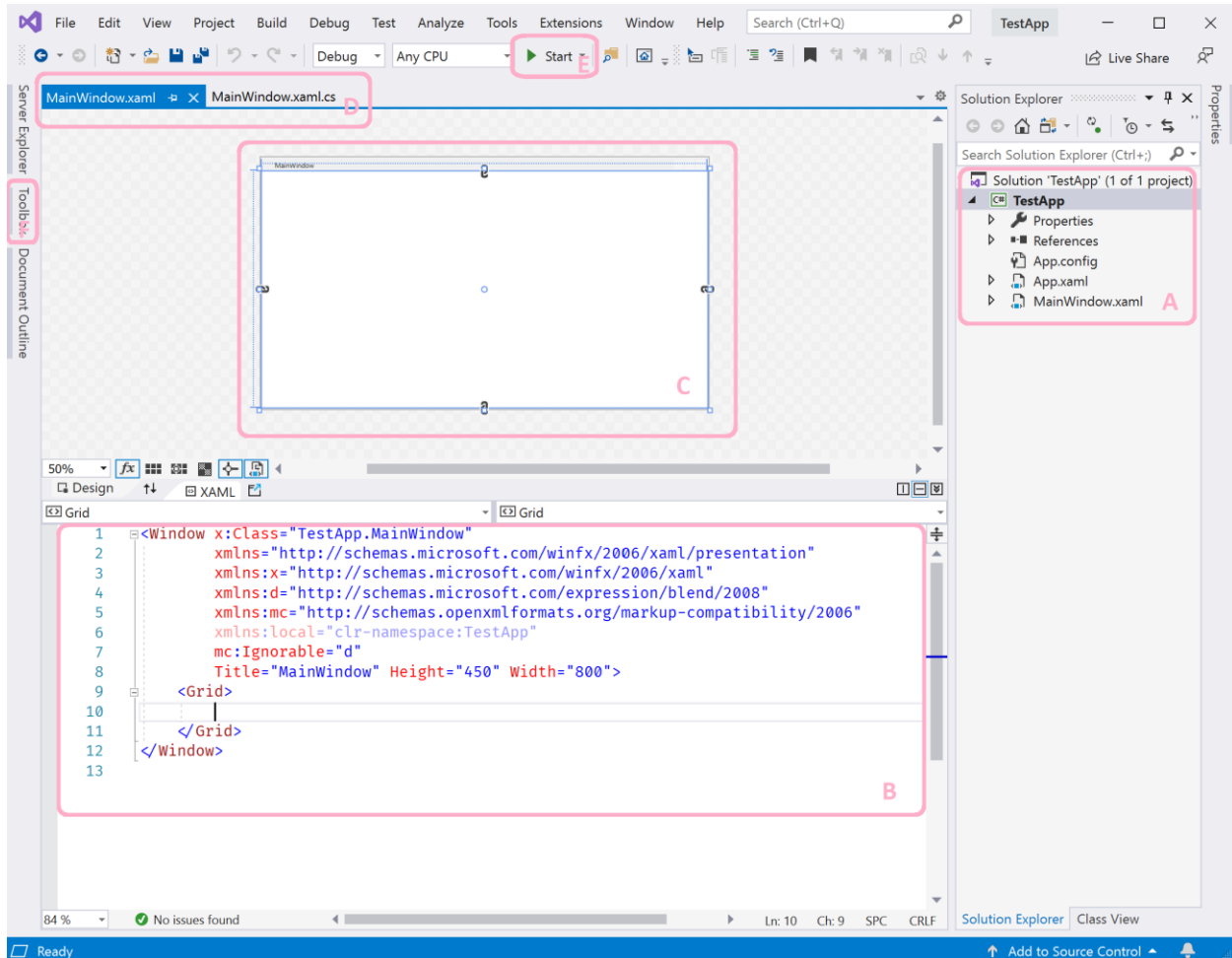
There are two rules to naming projects. The first rule is that there should be no spaces or special characters in the text, just the letters A-z, a-z and the numbers 0-9. The name should start with a capital and if, as in my example, the name is two words, capitalize the second word. This is known as pascal case writing because of the humps the capital letters make. In addition to PascalCase, there is camelCase, snake_case, and kebab-case. Snake case was popular a while ago, but programmers have mostly switched to PascalCase and camelCase. HTML uses kebab-case in its naming conventions.

The second rule is to name the project for what it does, not for why you are doing it. For example, if your first assignment as a programmer working for some company was to create an expense reporting application, an appropriate name would not be 'MyFirstApp'. Instead it would be 'ExpenseReport', or something to that effect. Try to keep your application name relevant to the problem being solved.

The other fields may be changed, but there is generally no reason to do so. However, do note the location. This is where the project's files are stored. The solution name and the project name will be the same – at least for small projects. For larger efforts, the solution name may reflect an umbrella title for a group of projects. The framework version is set to the latest version. Unless you have some specific need to support older versions of Windows, there should be no reason to change this.

A tour around your project

Now that you have decided on a name, click the Create button. After some time, you should be presented with a screen showing your project and its components. Below, I have highlighted the most interesting bits of this window.



The area labeled A shows you the files and folders associated with this project. By default, Visual Studio adds:

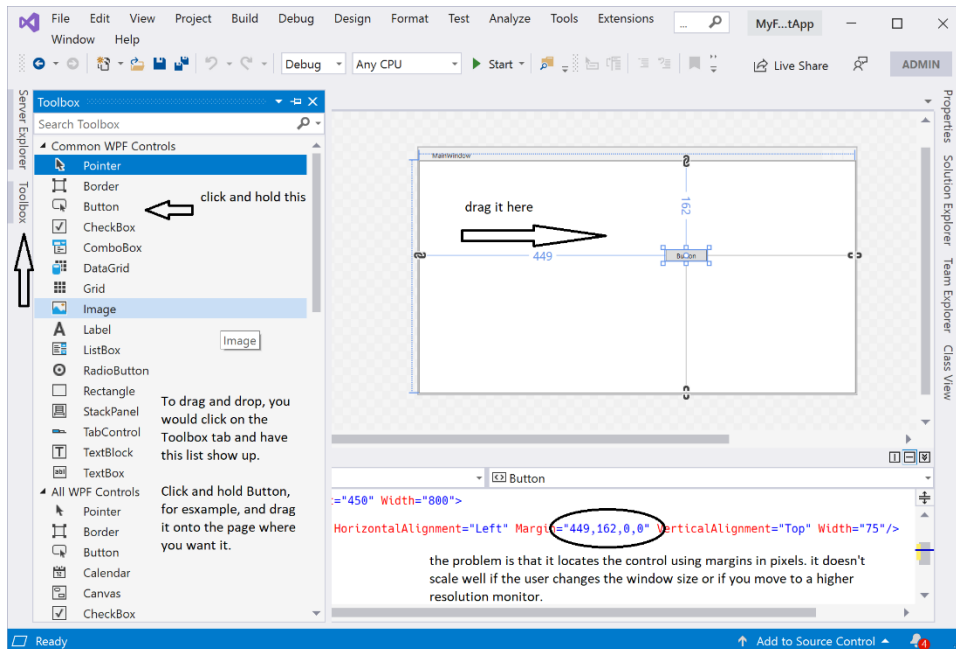
- Properties – leave this alone
- References – the parts of .NET that VS thinks we are likely to be using
- App.config – put configuration information here (we do not have any)
- App.xaml.cs – the application window (auto generated, and should be left alone)
- MainWindow.xaml – this is where we describe how our application should look
- MainWindow.xaml.cs – this is where we describe how our application should behave

The area labeled B is where you can modify the contents of MainWindow.xaml. The text that is there now should not be removed, but we can modify Title, Height and Width to change the window. If you do, notice that the area C also changes. This is the visual representation of any changes made in area B. In addition, any changes you make to C are also reflected on B. Use your mouse to click and drag one of

the hollow white boxes. Notice that when you let go, area B reflects the new state. As you approach a hollow white box, notice the cursor changes to a curve with two arrows. This allows you to rotate the object to any angle. Do that and let go. Notice all the text that got added to area B to represent that!

When you are done playing, type Ctrl-Z a few times. This will revert each of your changes. If you go too far back, type Ctrl-Y to put a change back. I show you this, so you will not worry about breaking something. You can always go back to a previous state, so playing is OK.

If you click on Toolbox (Area F) it will pop up a dialog showing you a list of prebuilt WPF controls. Click and drag one to the window (Area C) and release it somewhere on the surface.




Unless you are lucky, you will end up with some odd margin values:

`Margin="449,162,0,0"`

to describe the location in Area B. That kind of result makes it difficult to get things to align properly as we add more controls. As well, the width setting fixes the width of the button at 75 pixels, which means that your button will not be responsive to screen size changes. Although it is possible to drag and drop your way to an application, I will show you a cleaner way later. However, drag and drop does come in handy if you cannot remember the name of the control you want.

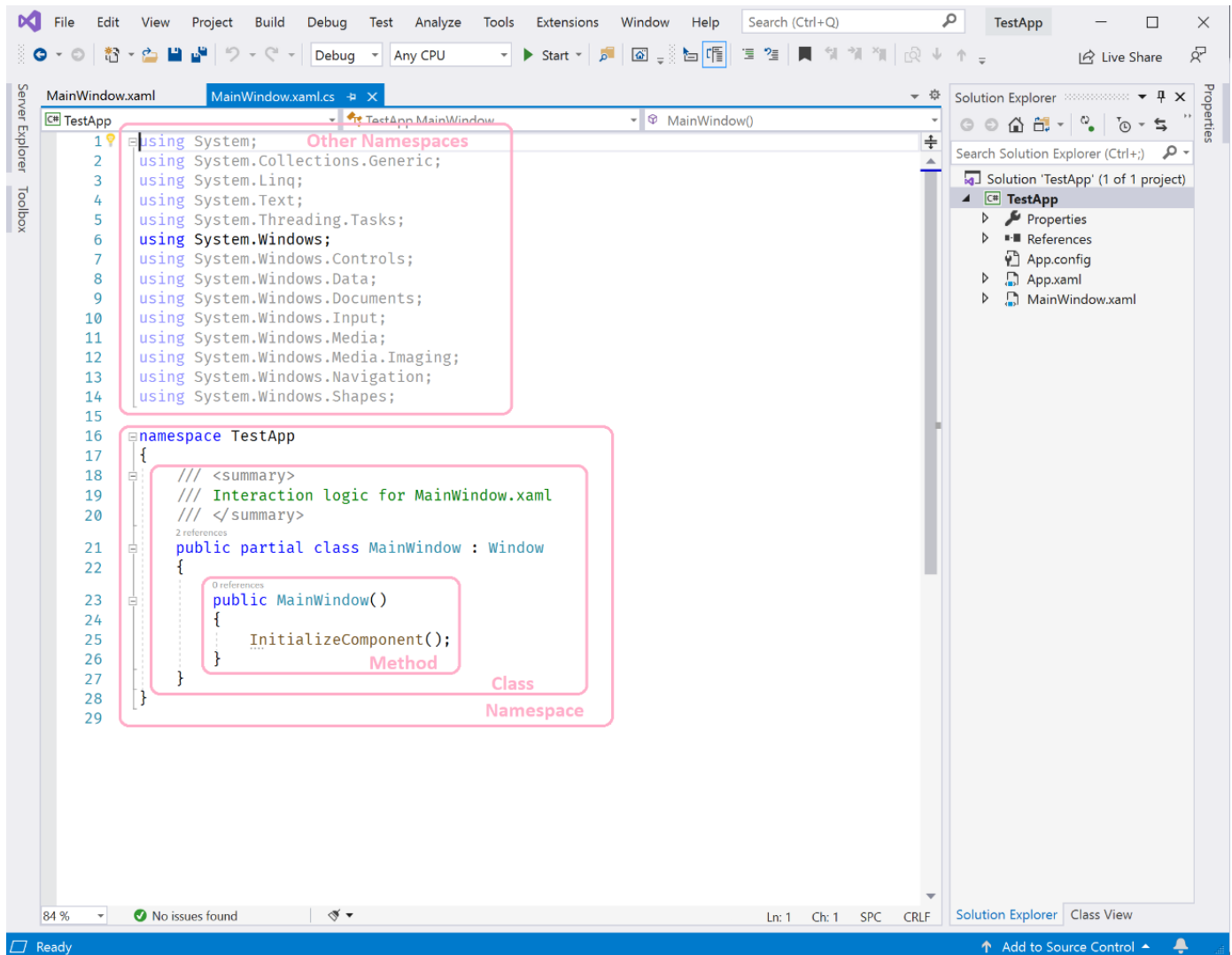
The area labeled D shows a set of tabs – one for every open document. The highlighted tab indicates which document is being shown, in this case MainWindow.xaml. Clicking on the other tab will bring the other file's contents up. As well, if you hover the cursor over one of the tabs and wait a bit, it will show you the full path to that file. So, if you forget where visual studio put your project, you can check again by hovering.

Finally, on to the Start button in area E. Click this now to try out your app. Or, as a shortcut, press F5. A few seconds after clicking Start, you should see a window appear with MainWindow as its title and little else – unless you added things while you were playing. The window does not do much but check out all

the things it does do. An icon representing your app appears in the taskbar. You can resize the window by grabbing a corner, you can minimize and maximize it, and you can close it. All the basics of a Windows window are provided for you at no effort on your part. When you are done trying things out, you can close your application the normal way with the close button, or you can use the stop button in the set of the now visible application control icons . The other two buttons are pause and restart.

Notice that when you click Start, there is a brief period where the bottom bar indicates 'Build started' and 'Build succeeded'. This is the compiling process and is creating a .exe file located in the same directory as your source files, but in a subdirectory (normally bin\debug). There you will find <your project name>.exe. If you double click on this file, you can run your code outside of Visual Studio.

We will finish the tour by clicking on the other tab in area D and make the contents of MainWindow.xaml.cs visible. By convention, .cs is the extension given to files that contain C# code. The MainWindow.xaml.cs file starts off with enough code – comprised of statements – to get all the default behaviors to work. In C#, a statement is generally terminated by the semicolon (;) character. A statement can cross multiple lines, it is only the semicolon that terminates it.



The 'using' statements at the top of the file list the parts of the .NET Framework (specifically other Namespaces) that Visual Studio thinks you might use. The greyed ones are the ones you are currently not using.

When we want to group a series of statements into a single unit, we use the opening and closing curly brace characters {} to denote the start and end of the unit.

The outermost unit, the namespace, is usually given the same name as the project and groups all other units in your project together, even across files. If you go back and peek at MainWindow.xaml, you will see that it also claims to be in the same namespace, although there it is described using XML syntax:

```
xmlns:local="clr-namespace:TestApp"
```

For our purposes, the namespace assigned to our application is not particularly important, so it is possible to skip the next paragraph and still successfully work with the rest of this book. It only becomes important once you get into more complex projects.

If you remember, I mentioned that one of the other project types we could have built is a library project. Since a library is intended to be shared across other applications, we do not want to worry that a name we use in our application will conflict with names used in a library. Namespaces ensure that there is no conflict because the namespace becomes part of the name of that item. To see this in action, have another look at the using statements. Each using statement indicates that we want to access something from a different namespace, but we want to use only its name. It is like saying that we promise not to use one of their names. However, if it happens that we need to name something with the same name as something in another namespace, we can always say <namespace><dot><unitname> to specify which of the identically named things we actually are referring to.

Inside of the namespace block, Visual Studio has put a class block. This is the basic unit for organizing code in C#. In this case, this class definition has the text ': Window' at the end. That annotation means that our MainWindow class will 'inherit' all the behaviors of the 'Window' class. That is why, when we ran the application, we had all those built in behaviors of a window.

When we want to accomplish something, something needs to invoke a method. Methods are found in classes. Methods are the unit that contains instructions for the computer to execute. In this case, Visual Studio provides us with a method to initialize our window – populating it with the controls we defined in the XAML.

Displaying text in a label control

Now that we've had the tour, it is time to start customizing the application and have it do something useful. The first thing we need to do is to provide some instructions to the user of our application. We will do that by adding a label to the window.

To add a label, switch to the MainWindow.xaml tab. Once there, click to put the cursor in between <Grid> and </Grid> in Area B, the XAML editing area. Now begin typing the characters

```
<La
```

Notice the popup window that appears. It is suggesting Label by highlighting it in its list. If you press the enter key at this point, you will accept its suggestion, so do that. Now type

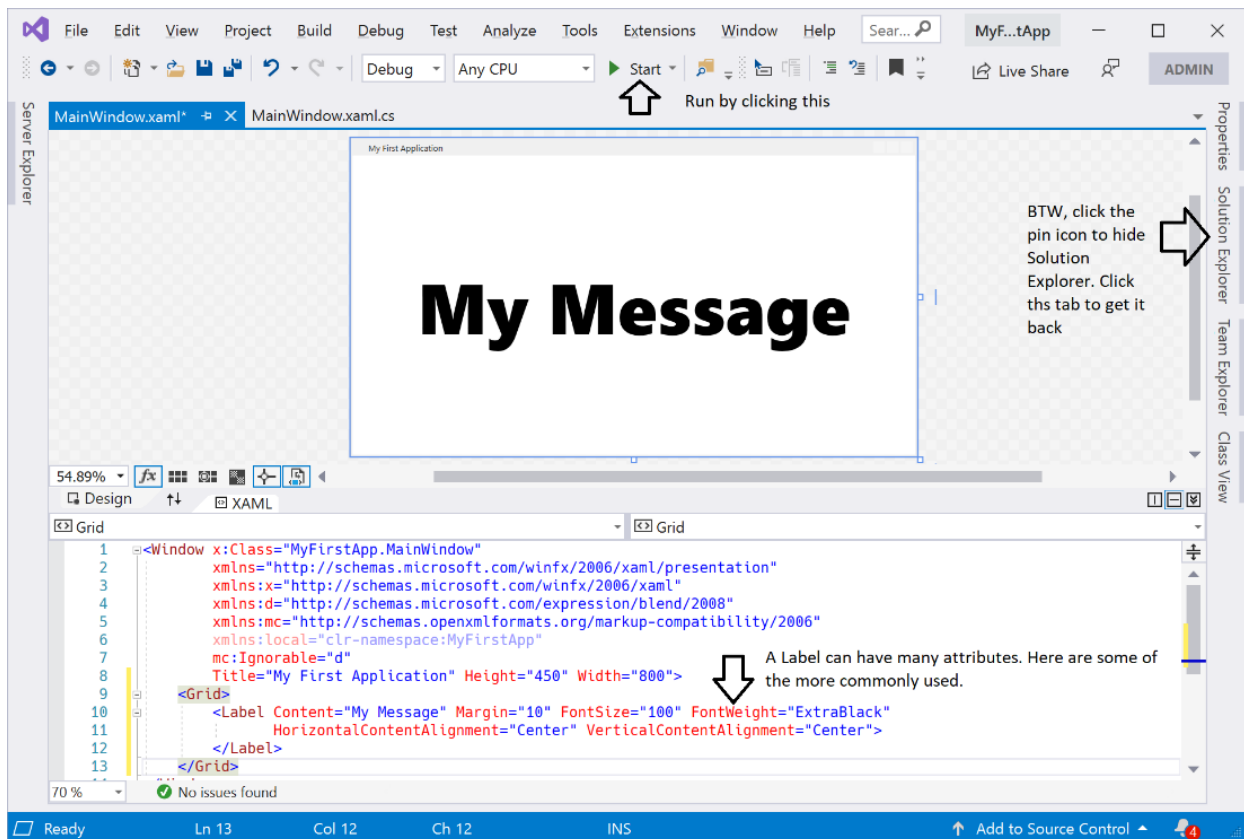
/ or >

Either will create a correctly formatted Label tag for you with the proper indentation.

Labels have a Content attribute that allows us to specify the content of the label. To set the content to be some text, place the cursor immediately after the <Label and type a space. Another popup should appear but this time, no suggestions are highlighted. However, Content should be at or near the top of the list. You can use the down arrow key to select it. Once it is selected, press Enter to add the attribute. Or, if you prefer, just start typing Con..., accept the auto suggestion. Now you should see a label tag that looks like

Content=""

with the cursor between the quote characters. You can now type your message inside the double quotes. When you do that you should end up with something that looks like:

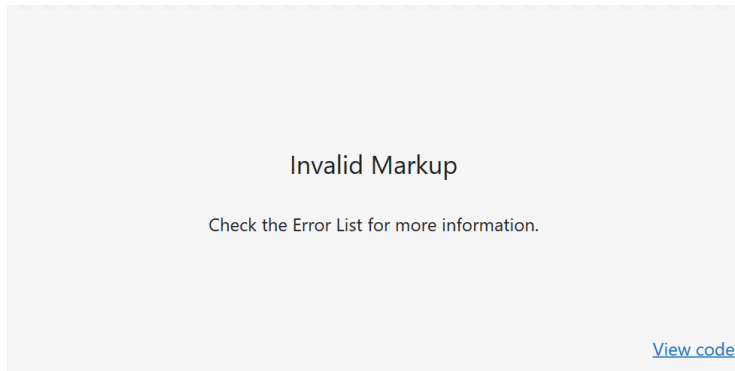


Notice that when the suggestion popup appears, it provides a list of options. Only some of those options are attributes you can set. You can tell which ones they are by the wrench icon at the left.

I have gone ahead and added more attributes. Add your own attributes or repeat mine with different values. Notice that as you add attributes, the upper area shows the results. Also notice that for some attributes, all you get is quotations and it is up to you to fill in contents. This is either because it is expecting freeform text (like Content) or a number, like FontSize. Others, like FontWeight have a fixed set of options. In that case, a popup will appear with the list of available options for that attribute.

Where possible, pick from the list. If you do happen to click away and lose the list, you can always get it back by typing Ctrl-Space.

If, at any time, you end up with an upper area that looks like this:



Just Ctrl-Z your way back to the point where that message disappears, and you see your layout again. You can use Ctrl-Y to put the offending text back and see what exactly the issue was. Usually the issue will be underlined in a wave blue or red line and, if you hover the cursor over that area, a text message will attempt to explain the issue.

The text of these explanations can seem cryptic, but the programmers behind Visual Studio are trying to be as precise as possible in explaining the issue so that two issues do not end up with the same explanatory text. That sometimes means it can be challenging to understand. But, once you get used to it, you will know exactly the issue based on the text.

Automatically resizing the text

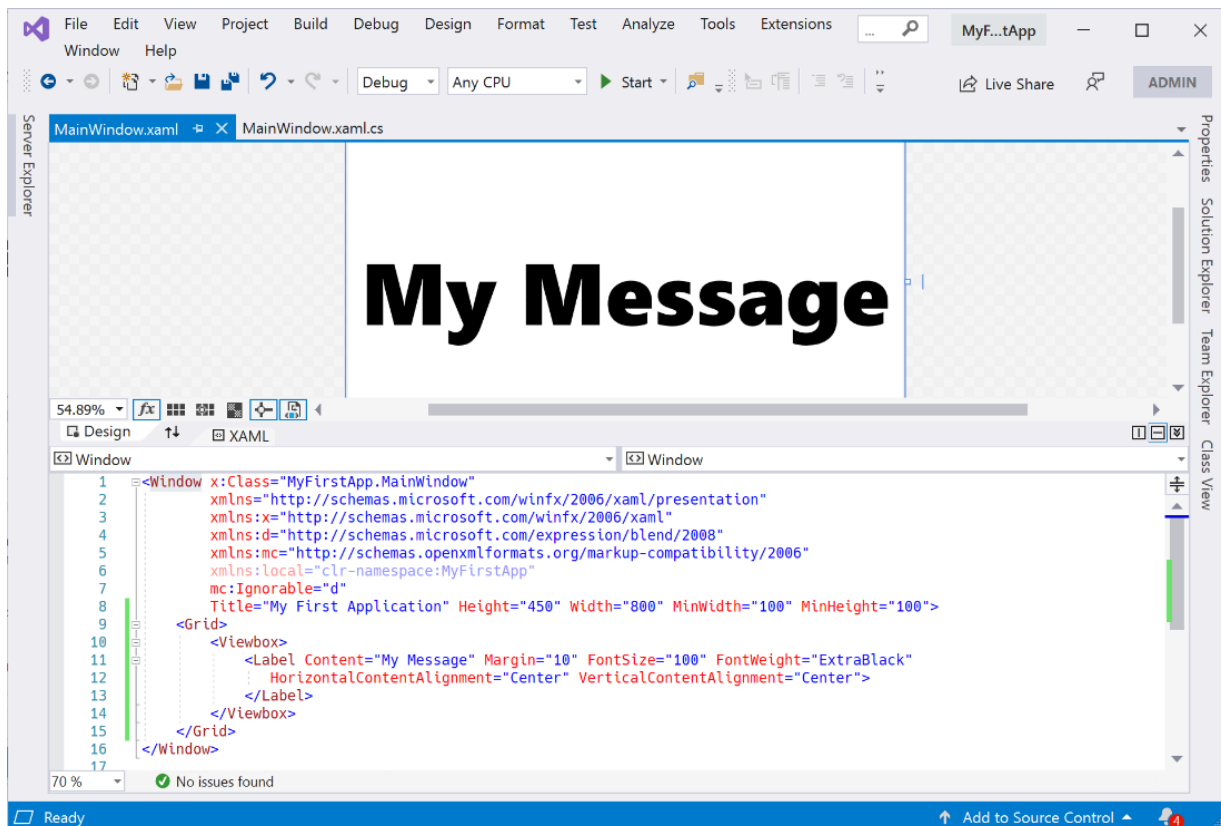
Run this latest version of your app with F5 or Start. Grab a corner and resize the window.

When you resize the window, you will notice that the text does not change size. We have set it to some fixed FontSize – even if you set nothing, you have implicitly set it to the default – and that is the only size it will ever be. However, many applications need to resize their text to work equally well on small and large screens.

To do this, we can use the Viewbox control. The Viewbox control resizes everything inside itself to the maximum size it can be yet still fit within the space given. To make the Label control resize as the window resizes, what we need to do is ‘wrap’ the Label control with a Viewbox control. If you have not already closed the app, do not. In Visual Studio with the MainWindow.xaml tab selected, put the cursor after <Grid>. Type Enter and the characters

<Vi

At this point, the suggestion should be highlighting Viewbox. Use Enter to select it and type >. This will close the tag with a separate </Viewbox>. We want that closing tag, but not until after the Label tag is closed. So, with the cursor in between the >> characters, hold down the shift key and use the right arrow to highlight the entire end tag. Now type Ctrl-X. That cuts the selected text. Use the arrow keys to navigate to immediately after the </Label> closing tag and then type Ctrl-V. That pastes the selected text into its new home. While you are here, try the Home and End keys to see what they do. They will help you navigate around your application without having to take your fingers off the keyboard.



Notice that if the app is still running, changes we make to the XAML are immediately reflected in the running application. This helps you iterate over the look of your app quickly.

Because resizing also means it will shrink everything to fit, I set a minimum size for the window using the `MinWidth` and `MinHeight` attributes of the `Window` tag.

Now that you have played with `Viewbox` and have seen what it can do, remove it so the behaviors I demonstrate next will be more obvious.

Displaying a button

At this point, we have displayed something to the user. But to make it a useful app, we need the user to interact with our app (other than to close it). The traditional way users interact with applications is through buttons. So, add one just like we added a `Label`. Place the cursor in the editing area of the XAML window so it is positioned inside the `Grid` tags, but outside of the `Label` tags – usually on its own line. Start typing

<But

Accept the suggestion of `Button` and close the tag. Like labels, buttons have a `Content` attribute. Like labels, you can assign text to the `Content` attribute. For example, "Press me". If you look up to the display area, you can see that our window is now filled a grey box (the button) and our label is missing. Label is not actually gone. It is underneath the button. That is, if you put the button tag after the label tag in the XAML. If you put it before, the label text now has a grey background and the tiny words 'Press me' can just be made out. This is known as Z-Order (as in, the Z axis of a graph). Items are placed in

order of appearance in the XAML, so the last things are layered on top of the first. The reason we can see the button behind the label is that its background is transparent. However, a button's background is not transparent, so we cannot see anything that is behind it.

Grid rows and columns

Although having to components layer on top of one another can be a cool effect, we really do not want that in this case. Instead, we want to separate the two items into their own spaces. To do that, we are going to use the Grid more fully and divide it up into rows and columns. We do that by defining the number of rows and columns for our grid by defining rows and columns inside the Grid, like this:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Button Content="Press Me" />
  <Label Content="Some text" />
</Grid>
```

If you added this set of definitions to your XAML, you have noticed that it does not change anything. That is because I have defined one row and one column – exactly what was already there. To add a second row, we need to add the tag `<RowDefinition Height="*" />` one more time. To add a second column, we add the tag `<ColumnDefinition Width="*" />` one more time. Now a check of the preview window shows that we have indeed changed things. Our elements are now both in the top left corner and three other equally sized cells are empty. We still need to specify something to move either the button or the label out of that cell. The way we do this is by adding a `Grid.Row` and a `Grid.Column` attribute to the Button and Label XAML, like this:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="2*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Label Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
    HorizontalContentAlignment="Center"
    VerticalContentAlignment="Center"
    Margin="10" FontSize="150" Content="Blah Blah" />
  <Button Grid.Row="1" Grid.Column="0"
    Margin="10" FontSize="50" Content="Press Me" />
</Grid>
```

and assigning a different row/column combination to each item. The top left quadrant is row 0, column 0. One step to the right is row 0, column 1. One step down is row 1, column 0. When we specify nothing, `Grid.Row` and `Grid.Column` are assigned zero. Note two other changes I have made. The first `RowDefinition` now has a height of `2*`. The `*` means to allocate the available space equally to all rows. By putting a number in front of the `*`, we are specifying the fraction of the total to assign to the row. It is

calculated by adding up all the values in front of the * (1 is implied if it is not explicitly written). In our case, that is $2 + 1 = 3$ as our denominator. So, the top row gets two thirds and the bottom row gets one third of the available space. It is also possible to specify a row height or column width in pixels. To do that, use a number without the *.

Second thing to notice is the `Grid.ColumnSpan` attribute in the `Label` definition. Without it, the label is only allowed to occupy the single top left cell. But add this attribute, and the `Label` can stretch across both cells in the top row. This comes in handy when you need a finer grid to place all your elements, but an element or two require a bit more room. There is also a `Grid.RowSpan` attribute available if the element needs to stretch across rows.

Using the editor to *finally* write some C# code

Clicking a button

We now have a reasonably good-looking user interface (UI), but the app still does not do anything. It is finally time to add a behavior, or as one would normally say in Windows – handle an event.

The most basic event in a Windows system is the click of a button. Since we already have a button, we will add a handler to that button for the click event. This we do with a new attribute. To add it, repeat what you did to add other attributes, but this time the click and check the contents of the popup. You should see `Click` listed. It will have a lightning bolt beside it. This symbol indicates an event. If you scroll through, you will find there are many other events, but will stick to the easy one for now.

Select `Click` from the list. This should give you a new popup with only one element in, already highlighted: `<New Event Handler>`. Type the enter key to select that. It has filled the value of the `Click` attribute as `"Button_Click"`. If you set up a `Name` attribute for the button, then it will instead say `<Name of button>_Click` as the value. Since we only have one button, `Button_Click` is a reasonable name, but in the future, we will have several buttons, so we should always give buttons a name with the `Name` attribute.

There has been another modification besides the attribute change. Notice that Visual Studio now marks `MainWindow.xaml.cs` as changed (that is what the * after the name means). To see what has changed, click on that tab. You can see there is now a new method block called `Button_Click` added to our class.

Any C# statements we add in between the `{}` characters of the `Button_Click` method cause something to happen when the user clicks the button. Statements are executed from top to bottom of a method, starting at its first line after the `{` and working its way down until it reaches the `}`. It then is done executing your code and waits for another event to occur.

MessageBox

The first statement we will write will cause a message to appear to the user. We can use the built-in `MessageBox` class to do that. Position your cursor between the `{` and `}`. The cursor should be on its own line and indented four spaces inside of the `{}`. Type

`Mess`

and it will suggest a few things, but `MessageBox` is the one we want, so select it.

Although this is correct so far (you can tell because the word changed color), Visual Studio will draw a red wavy line under the word since it is not yet a complete statement. Typically, a class can do one of several actions, so we need to specify an action for `MessageBox` to do. The actions are accessible by typing the period character (.) immediately after the word `MessageBox`. Now a new list of options is displayed. The three you should see are: `Equals`, `ReferenceEquals`, and `Show`. The first two do not sound like the name of an action (actions have the little cube icon beside them) that would cause something to be displayed. However, `Show` seems like just the thing. Select that. Now you should have the red squiggly line trailing after the word `Show`. If you take your cursor and hover it over the red squiggly, you should see something like:



and this is Visual Studio trying to tell you the proper way to use `Show` as well as describing what it does. The important thing to notice is that the text `MessageBox.Show` is immediately followed by an opening round bracket character (and then a bunch of other stuff. See the place where it says (+11 overloads)? That means there are 12 ways to make the `Show` action do something. If we type the (, we will be able to see those 12 choices. So, do that. You should now see:



Notice that as soon as you typed the (, Visual Studio added the) and put the cursor in between. Now the description has changed somewhat and the list of things between the () in the description has shrunk considerably – in fact, it is down to one thing. And, reading further into the description, this one thing ‘`messageBoxText`’ specifies the text to display. Cool. We just need to figure out what a string is, and we can display a message.

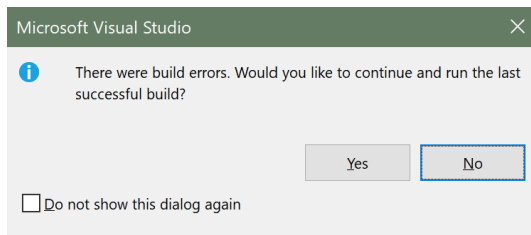
We have already used strings but without naming them as such. A string in C# is anything enclosed in double quotes if it does not extend over multiple lines. So, the simplest resolution to our requirement to display a message is to write something like:

```
0 references
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("hello");
}
```

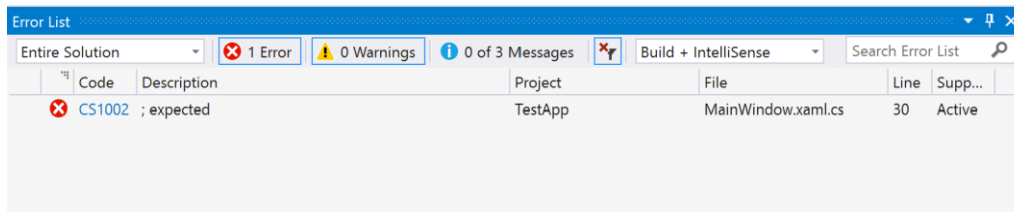
Notice the trailing semicolon. C# statements all end with that character. Without it, there would still be a red squiggly to let you know it had been forgotten. Also notice the color coding of classes, methods, and strings. Many times, you can spot an issue by noticing that the colors are not quite right.

Now that have a valid statement in our button click event handling method, we can try it out. Use F5 or Start to run the app.

If something is not right, you will get a message box popping up that looks like this:



Always press No in response to this dialog. You want to know what went wrong. Visual Studio will tell you in a new Error List window that looks like this:



In my case, I forgot a semicolon (since the description is ; expected) in the file MainWindow.xaml.cs on line 30. I can go there, fix that, and try to run again. The errors will be displayed from top to bottom of the file. You should fix them in that order because things like missing semicolons can cause a cascade of errors as Visual Studio attempts to understand and is confused by your code. In a complex application, fixing the first error of 100 can sometimes make the other 99 go away.

Now that everything is correct and running, press the button and you should see a message box saying hello.



Congratulations! You have created your very first working application.

Messages in Labels

Using a MessageBox is fine, but if you think about apps and websites you have interacted with lately, you probably realize that there are very few message boxes being used. Instead, an area in the window will get updated with some text. In the case of XAML, we can accomplish this by updating the content attribute of a label.

First thing we need to do is go back to the XAML editor and give the label a Name attribute. I will call mine 'MessageLabel'. We give the label a name because that way we can refer to the label in the C# code using this name. What that means, then, is that if I add the statement:

```
MessageLabel.Content = "you pressed me";
```

to the `Button_Click` method, running the program and clicking the button will display the message box and change the Content of the label. The order that two things happen depends on the order of the statements. If you put the new statement after the message box method invocation, then it waits until you close the message box before making the label content change.

Looking at the statement a bit more closely, we can see a couple of interesting things. First, in the XAML, we set the `Content` attribute. In C#, we are setting the `Content` property. The two `Content` things both refer to the same place in memory, but we use different language to describe them depending on which language we are using. Second, `MessageLabel` must be like `MessageBox` because we can use the period to access its properties. Third, we used an equal sign in a statement. The `=` means I want to assign the value of the thing on the right to the thing on the left. If you recall the discussion about working memory, then this shows you that principle in action. The string (the text enclosed in quotes) takes up some amount of memory. The `MessageLabel` also takes up some amount of memory. We know that we can set all sorts of properties for `MessageLabel`, so there must be memory set aside to hold that information. When we do the assignment with the equals sign, we are telling the computer to copy the contents of one block of memory and put it into another block of memory – replacing what was previously there.

Run this version and clicking the button changes the text. But only once, and then no more changes occur.

Comments

It is generally useful to leave notes to yourself or others that might have to deal with your code in the future to explain what you were trying to do. In programming terms, we call these comments. In C# code, comments can look like this with a double forward slash to start with everything else on the line considered part of the comment:

```
//this is a C# comment
```

Or this:

```
/*  
  this is a multiline  
  comment  
*/
```

With everything in between the `/*` and the `*/` considered a comment regardless of the number of lines enclosed.

In XAML, comments look like this:

```
<!--  
  this is a multiline comment  
-->
```

With everything between the `<!--` and the `-->` considered a comment.

The important thing about comments is that the compiler will ignore everything that is marked as a comment. Which means you can write anything you like in there, including other statements that you do not want the compiler to process. Typically, you use comments to explain why you are doing something, or the details of some particularly tricky bit of logic. I like to say that you need to explain things to future you because that person will not remember all the things that current you knows.

I will use comments throughout my example code to explain why I am doing certain things. Hopefully, from that you can get an idea of what your comments should contain.

Exercise 1

Create a new project. Place a label at the top of the form and, below that, two buttons side by side. One button, labeled “press me”, should change the label to contain the text “you pressed me”. The second button, labeled “press me again”, should change the label to contain the text “you pressed me again”. Run the code and see the label toggle back and forth between the two strings as you click each button in turn. Once you have done that, compare your code with the project in the Exercise1 folder.

Please note, there is not a single ‘correct’ way to solve a problem, there are only ways that work. If your version works and you are happy with the way it looks, then that is a success.

Adding a bit more style to our app

To this point, we have figured out some basic user interface elements: buttons and labels. That will get us a fair way down the road to a useful application. If you think about most applications and web sites you interact with, that is what they mostly consist of. There is one other very common element, and that is the display of images.

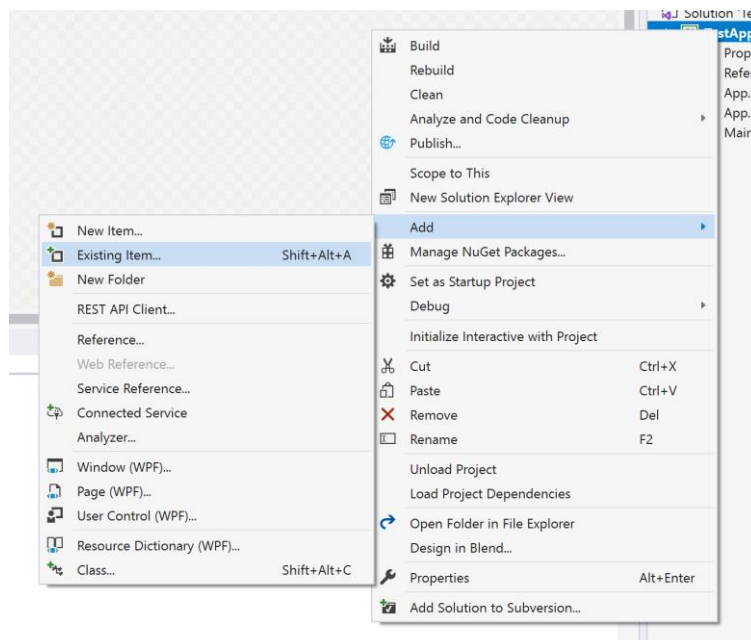
Displaying an image

We can use an image in many ways. We can set the background of our window to an image. We can put an image into the foreground. Or we can add an image to a button or to a label.

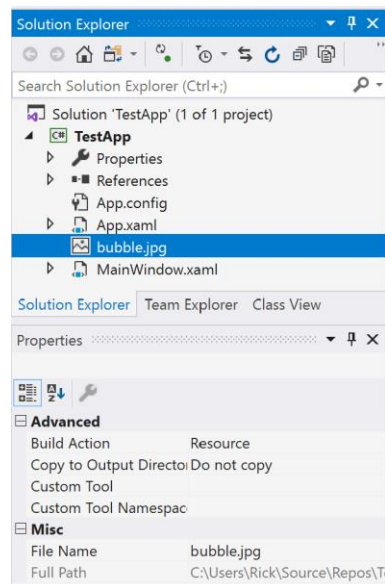
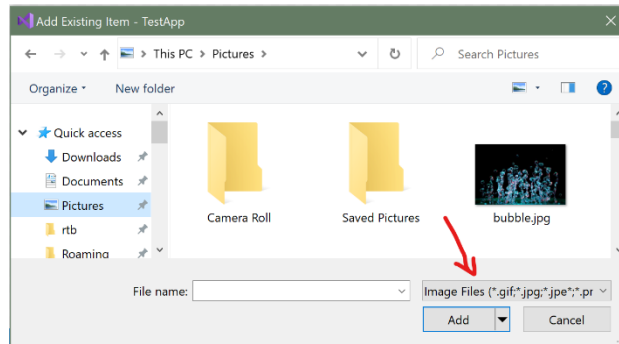
To set the background of a window, you need to add this code in the XAML:

```
<Window.Background>
    <ImageBrush ImageSource="C:\Users\Rick\Pictures\bubble.jpg"/>
</Window.Background>
```

somewhere between <Window> and </Window>. The problem with this approach, however, is that I have had to specify exactly where to find this jpg file. Of course, it will not be in the same place on your computer and that will be a problem if I want to have you run my application on your computer. One way to solve this problem is to bring the image into our project. To do that, right click on the project name in the 'Solution Explorer' window and select 'Add' then 'Existing item...'



Now you should see a standard Windows File Open dialog where you can pick the file you want to add to the project.



Generally, this is a good thing. However, if you want to allow users (or perhaps your program) to switch out images, then we will want to change the 'Build Action' to 'Content' and the 'Copy to Output Directory' option to 'Copy if newer'. In this case, the file is kept separate and can be replaced later without having to rebuild your application. In any case, the option you chose is immaterial to the remaining discussion.

Now that the Properties window is open, notice that when you click a line in the XAML, the properties window changes to show you the current for and other available attributes for that element. I won't be referencing that interface again, but you may find it a handy guide to what is possible.

```
<Window.Background>  
    <ImageBrush ImageSource="bubble.jpg"/>  
</Window.Background>
```

One thing we should do if we have many image files is to create a folder to hold them. To do that, right click on the Project again, select 'Add...' and this time select 'New Folder'. Rename the folder to 'Images' (right click on the folder and select the 'Rename' menu option). Now drag and drop the image file into the folder. This keeps our project more organized. As you create larger projects, you will make more use of folders to group like things together to make them easier to find.

To allow the app to find the new location for the image file, we need to make one last adjustment.

```
<Window.Background>
  <ImageBrush ImageSource="images\bubble.jpg"/>
</Window.Background>
```

I have added what is called a relative path to the file name. What we first used was called an absolute path – since it specified the entire path from the drive letter all the way to the image name. A relative path causes the app to start looking for the file in the directory where the .exe file resides and then follow the folder path information from there.

In addition to using images in the background of our window, we can also pick colors. One of the more interesting effects is the RadialGradientBrush. For example, this:

```
<RadialGradientBrush GradientOrigin="0.25,0" Center="-0.25,0" RadiusX="1" RadiusY="1">
  <RadialGradientBrush.GradientStops>
    <GradientStop Color="GreenYellow" Offset="0.1" />
    <GradientStop Color="Yellow" Offset="0.25" />
    <GradientStop Color="Red" Offset="1.5" />
  </RadialGradientBrush.GradientStops>
</RadialGradientBrush>
```

Produces a background like this:



Note that all the values for all the properties are floating point numbers where 0.0, 0.0 means the top left corner and 1.0, 1.0 means the bottom right corner. Which means that my center at -0.25,0.0 is outside of the window to the left at ¼ of the window width and the Red GradientStop is outside of the window to the right. You can think of these as percentages, where 1.0 means 100%. This makes specifying something that will fill the window easier, since the numbers do not change regardless of the window size in pixels.

The other thing I am sure you noticed in the previous image is that I have changed the look of the label and the button. If you tried to use the bubble.jpg image as the background, you would have immediately noticed that the text was no longer readable. To make the label and button clearly understandable on a dark background, I modified the label and button like this:

```
<Label Name="MessageLabel" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
  HorizontalContentAlignment="Center" VerticalContentAlignment="Center" Margin="10"
  FontSize="100" FontFamily="Arial Rounded MT" Foreground="WhiteSmoke"
  Content="Blah Blah"/>
```

```
<Button Grid.Row="1" Grid.Column="0" Click="Button_Click"
        Margin="10" FontSize="50" FontFamily="Magneto"
        Foreground="WhiteSmoke" Background="Transparent"
        BorderBrush="WhiteSmoke" BorderThickness="3"
        Content="Press Me"/>
```

For the label, changing the foreground color is all that is necessary. The button, on the other hand, requires a bit more. In addition to changing the foreground color to white, I made the background transparent. By doing this, our cool background color now shows instead of the boring grey default button face color. But it also makes the button border invisible. To fix that, I added a border color and thickness to make the outline of the button visible again. And, of course, I picked fonts that I think match the happy feeling of the background.

You should take some time to play around with the assorted color, font, and background options. One word of warning, though. Ensure that readability and usability come first in your design decisions. Readability usually comes down to contrast, although script style fonts can complicate things and should be used only sparingly and as large text. You can use a tool like this

<https://webaim.org/resources/contrastchecker/> to make sure the contrast between your text and background is good enough.

Also think about color blind people when you pick colors. A good way to check is to convert the image to



grey scale with a tool like this one <https://onlineimagetools.com/grayscale-image>. If the image is still readable as grey scale, then it will be readable regardless of the type of color blindness a user might have. Notice how my background is almost the same shade of grey throughout. What it means is that there will always be high contrast between my background and the text.

If you do use an image as a background, you might find yourself wanting to use an image with areas that are bright and other areas that are dark. This poses a problem for picking a font color, since no single color will have high contrast against both light and dark areas. A simple way to deal with this is to set the opacity of the background, like this:

```
<RadialGradientBrush Opacity=".3" GradientOrigin="0.25,0" ... >
```

Like the ranges, Opacity runs from 0.0 to 1.0 where zero means completely transparent and 1.0 means completely opaque. If you specify a low opacity number, then the white colored background behind the background will bleed through and will soften the differences between the light and dark areas of your image. Then a single font color selection is again possible.

Foreground Images

To put images in the foreground, we can use an <Image> element, like so:

```
<Image Grid.Row="1" Grid.Column="1" Margin="20"
```

```
Stretch="Fill" Source="images/bubble.jpg"/>
```

This will put the bubble.jpg image into the cell at 1,1 – the bottom right corner of our 2x2 grid. Also notice the ‘Stretch’ attribute. This has 4 options: ‘None’ shows nothing, and ‘Fill’ fills the available space with image, ignoring aspect ratio. ‘UniformToFill’ fills the available space but does pay attention to aspect ratio, meaning that some part of the image will be cut off. ‘Uniform’ attempts to fill the space as best it can while still maintaining aspect ratio, potentially leaving empty space around the image. Whatever we do, however, the image will look a bit odd, like someone took a punch to our background and created a peephole through which we can see this image. To fix that illusion, we can make the image seem to float above the background using a drop shadow. A straightforward way to create one is to put a rectangle in the same grid cell, like this:

```
<Rectangle Grid.Row="1" Grid.Column="1" Margin="30,30,10,10"
Fill="Black" Opacity=".5"/>
```

I have set the margins so that the left and top are hidden behind the image and the right and bottom are sticking out by 10 pixels. The larger the apparent shadow, the higher above the background the image seems to float. Of course, do not forget about Z ordering. If you want the rectangle to appear to be a drop shadow, it needs to be drawn behind the image, which means it needs to be defined first in the XAML.

Usually, we would use the built-in drop shadow effect, but we will learn about that and other built in effects later in the book.

The one thing an Image element cannot do is accept clicks. If we want to let the user click on an image to cause an action, then we need to combine the image element and the button element. Fortunately, the Content attribute of the button element can be more than just text. If we define it as an attribute, then it can only be text, but we can also define the content in the body of the button XAML element, like so:

```
<Button Grid.Row="1" Grid.Column="1" Name="Image" Click="Image_Click"
Margin="10" Background="Transparent">
  <Grid>
    <Rectangle Margin="20,20,0,0"
      Fill="Black" Opacity=".5"/>
    <Image Margin="10"
      Stretch="UniformToFill" Source="images/bubble.jpg"/>
  </Grid>
</Button>
```

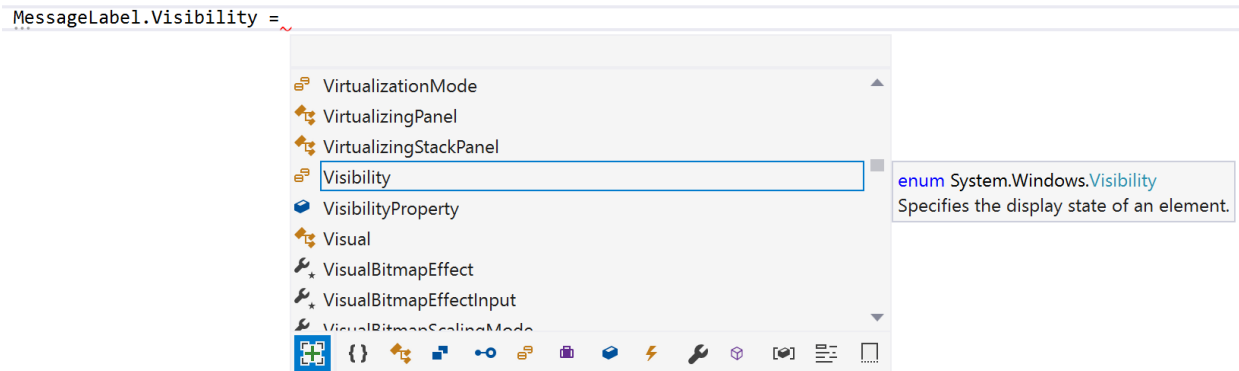
What has happened now is that I have taken the Rectangle and Image elements and wrapped them inside their own grid. Then I have taken that grid and put it inside the button element. I had to wrap the two elements inside a 1x1 dimensional grid because the button content can have at most one top level element. What happens inside that element is up to the rules of that specific element.

Even after all this, we still have the default powder blue behavior when the mouse hovers over the buttons. We can take care of that too, but not quite yet. We will save that for later in the book.

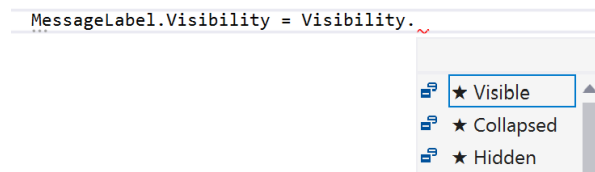
Showing and hiding controls

We have been setting control attributes using XAML for the most part. However, just like with the Content property, other attributes can be controlled through code, including one of the more common – visibility. To set the visibility of a control, use its ‘Visibility’ property in C#. You might expect that visibility

would either be true or false, but instead, when you type the = sign, Visual Studio suggests something else, an item called Visibility that is something called an enum – we will talk more about this later.



Select that item and type the period. You will be offered a list of three choices: Visible, Hidden, and Collapsed.



Visual Studio puts a star beside what it thinks are the most likely options given the context. Pick something appropriate and end the statement. For example, if you have two buttons, you might have one of them make an element Visible and the other button make it Hidden.

The reason for collapsed and hidden as two options is to determine what happens to the space occupied by the control. If hidden, the space remains empty. If collapsed, then any elements that can, will flow in to occupy the space. With what we have learned so far, this distinction has no meaning. However, there are container elements that we will discuss later where this is relevant.

Code Cleanup

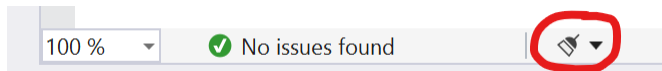
As someone just starting out as a programmer, you will have more senior developers as supervisors and mentors. The best way to make a favorable impression is to make your code neat and tidy. Bugs are inevitable and will find their way into everyone's code. But you have complete control over the style of your code. If you were to come to me for help and presented me with this "style":

```
0 references
public MainWindow()
{
    InitializeComponent();
}
```

The first thing I would tell you is to format your code properly and come back.

Most professional programming shops have a set of rules they follow for formatting. These are, at times, hotly debated. The tabs vs spaces (spaces, always spaces) argument has gone on as long as I have been programming. The question of how much indentation, (2 spaces or 4), where to locate the opening { (same line or next line), and multiple other details are endlessly debated. I do not much care for a specific style. What I do care about is consistency, both within your code and between your code and that of your co-workers. If I am your co-worker and you insist on being different – even if you are consistent – I, and everyone else on the team, will not be happy with you. That is because if all these decorative things common to the programming language are in the same places, they can be mentally tuned out and I can focus on the meaningful bits. If you distract me, then it becomes harder to extract those meaningful bits, which means I must work harder to do that. And I’m a lazy programmer, so I don’t want to work that hard.

Fortunately, there is a very easy way to clean up most of the mess. Visual Studio, like most developer tools, has a built-in formatting tool. To use it, find the broom icon at the bottom of the Visual Studio UI and click it.



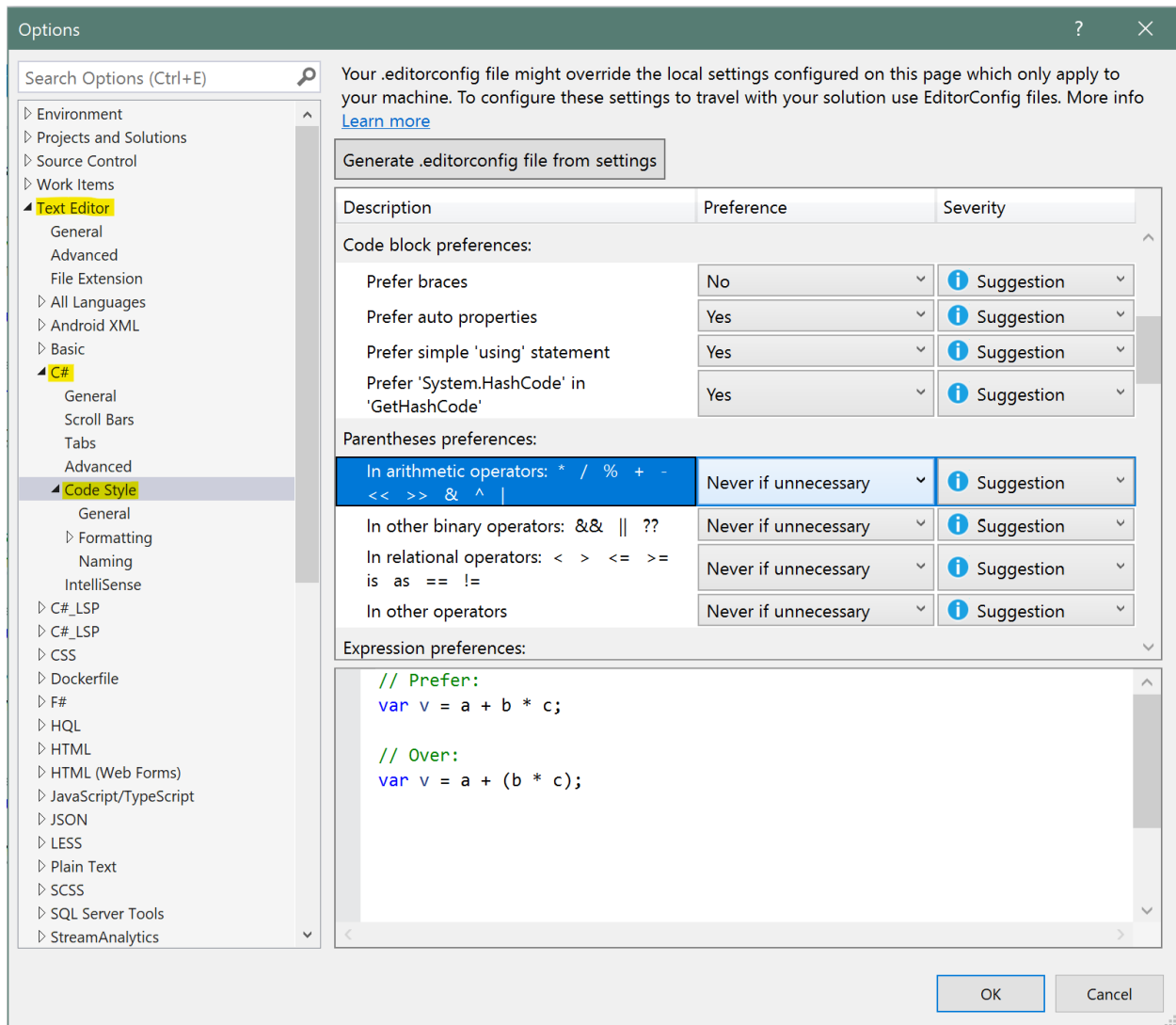
This fixes the alignment issues, which are crucially important. In more complex applications where multiple pairs of brackets at various levels of indentation are present, poor alignment can and will cause you to misread the code and not find an issue you are hunting. Get used to ensuring correct alignment now, while things are still simple.

I know I just said that I do not have a style in particular. However, one thing I do insist on is those unsightly extra blank lines must go so that your code looks like this:

```
0 references
public MainWindow()
{
    InitializeComponent();
}
```

When you are writing a document, extra spacing or a blank line separate paragraphs. And, of course, paragraphs indicate new thoughts. There is never a requirement for more than one blank line in a row. This is also true when you are writing code. A blank line – and I mean only one blank line at most – show the separation between thoughts. Use them only when you are attempting to communicate something to the reader.

Although most programmers accept the default formatting rules set in Visual Studio, it is possible to change them. Click Tools -> Options in the menu bar to get configuration settings pages.



Set it up as you prefer. To share this, use the “Generate .editorconfig file from settings” button and pass the file around. If Visual Studio finds it, it will use the file.

Be aware that, as far as I am concerned, messy code is a symptom of a messy mind, and a messy mind will not be the mind of a good programmer. If you want to make a good impression, your code needs to be organized and tidy. Bugs are inevitable and everyone’s code will have them, but you have complete control of the formatting so judging you on that basis should be expected.

Years ago, I was working with a multidisciplinary team of programmers and hardware designers to develop what was, at the time, a sophisticated video effects device. We had spent almost two years and many thousands of dollars developing the technology and building a prototype that consisted of two black metal boxes connected with a cable. Eventually it was time to present the results of all our work to management. We brought them in and went through a forty-five-minute demonstration of its capabilities. Once done, we opened the floor to questions. Now, these managers did not understand the technology, but they were responsible for green lighting a release to manufacturing. The first question

we got was: “why did you use silver screws?”. For the most part, people use a shorthand for determining the success of some piece of work and, fair or not, judge all of it based on that shorthand.

Exercise 2

Create an application that has two image controls and four buttons on a form. Put one of two images on each button. Initially, both image controls should be hidden. The first pair of buttons will cause the image control to toggle between the two images by making one of the image controls visible and changing its source property. The second pair of buttons will toggle between the two images by alternating visibilities of the two image controls. Once you have done that, compare your code with the project in the Exercise2 folder.

Using memory for our own purposes

Variables

So far, we have been using the content property of label and button controls to store any information that is specific to us. In most programs, however, we will need to store data temporarily in our program outside of the properties of controls. We call these blocks of memory to store our data *variables*. And, because we will usually want to keep track of multiple things, we need to name each variable. In addition to naming the variable, we need to tell C# what kind of data to store in the variable. Some languages, like JavaScript, do not require this information. But since C# is a strongly typed language, it does.

The common simple data types in C# are:

- string – stores a set of characters or a special value called null (the *no value* value)
- int – stores numbers between -2,147,483,647 to 2,147,483,647 in 4 bytes of memory
- long – stores numbers between -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 in 8 bytes of memory
- decimal – stores numbers in approximately the range -7.9×10^{28} to 7.9×10^{28} (with a variable decimal point) in 16 bytes of memory
- float – stores numbers between $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ (with 6-9 digits of precision) in 4 bytes
- double – stores numbers between $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ (with 15-17 digits of precision) in 8 bytes
- bool – stores a boolean value (ie. True or False)

We can also have more complex data types, like DateTime, List, Dictionary, Control, Window, and your own custom data types.

We have already seen the string data type in action. We didn't give the string a name, but we did cause memory to be allocated for the block of text.

The simplest data type is the bool, as it can hold one of two states: true or false. We use this a lot for remembering user selections, which are usually a series of yes/no questions. However, the most common data type in use is int. Generally, we count an integer number of things and we usually don't count past 2 billion, so an int data type is often enough. There are, however, times we do want to count higher – like when we want to know how many unused bytes we have on our 500GB hard drives. In that case, the long data type is more than enough. For scientific calculations where integer math is not enough, the float and double data types are useful. Depending on the level of precision of your calculations, one of these two data types will provide the range required. However, they do suffer from rounding errors due to their precision, so the decimal data type is preferred when dealing with money as the range of values shown above can be represented without rounding errors. And we do not want rounding errors in our financial calculations. It can take a long time to work thru a set of calculations where the result is off by a penny.

The whole process of giving a type and a name to a block of memory is referred to as *declaring* a variable. So, for example, if we want to declare a variable that will hold an integer value, it looks like this:

```
int theVariable = 7;
```

The text 'int' specifies the data type and comes first. Next, 'theVariable' is the name we are giving to the location in memory that the system will assign to this variable. As we already know, the '=' is the assignment operator in C#. '7' is an integer number that will set the initial value stored in the memory that has been allocated. Finally, the ';' ends the statement – like it always does in C#. So, in total, we are assigning the number seven to be stored in the location labeled theVariable.

If we want to later change the contents of the variable, we can write another assignment without a data type:

```
theVariable = 4;
```

or, if we want to use the variable, we can assign it to another variable like this:

```
int otherVariable = theVariable + 1;
```

This statement takes the contents of 'theVariable' (which is currently 4), adds one to it and assigns the value 5 to a new variable 'otherVariable' that we just declared. In this way, we can combine and transform data – and that is the definition of a useful program.

In addition to the data types above, we can use var in our declaration, like:

```
var theVariable = 7;
```

This is not an additional data type but is a placeholder variable type that can be used if it is obvious what the data type of the variable should be. For example, in the above, theVariable is being assigned an integer value, so its data type is int. If, however, we wrote:

```
var theVariable = 7d;
```

Where the 7d means that this is a double value, then theVariable is a double. Or, writing:

```
var theVariable = 7f;  
var theVariable = 7m;
```

theVariable is set to be a float (f) or decimal (m) data type. In addition, once theVariable is declared, its type cannot be changed. It keeps its data type for its entire lifetime.

Constants

Constants are like variables, except that they can only be changed by you, the programmer, before running the code. We could just add non-changing elements directly into our code – like the assignment of the number 4 to theVariable above. However, this will make our code less readable by our future selves and, perhaps more importantly, by others. When you see a line of code like that, you wonder, why four? Why did they not add five or three? And now you must think about the code and try to determine if this is valid. As you are writing the code, it is obvious to you why you need to add four. But six months from now, after you have written many other programs and you need to revisit this one, will you still remember? I do not think you will.

Instead, it is good practice to use constants like this:

```
const float EARTH_GRAVITY_METRIC = 9.806f;
```

Now we know what the number 9.806 is supposed to represent and as we are reviewing the code, we can tell if the number is being used properly. Looking at this first example, you might think to yourself: “that’s such an obvious number, how could I mistake it for anything else?”. But we could also have a situation where we have need of the following values:

```
const int INCHES_IN_FOOT = 12;  
const int ONE_DOZEN = 12;  
const int TAX_RATE_PERCENT = 12;
```

Now, if you wrote 12 directly in your code for all of the occurrences of these values and the tax rate changes to 13%, you must now examine each 12 in your thousands of lines of code and decide which of the three possible meanings of 12 it has. If you used the constants in your code, then you only need to change this one place from 12 to 13 and your work is done. This is an example of what I mean when I say you need to be ‘long term lazy’. It costs you a tiny bit extra to define a constant now, but it will save you significantly more work if something needs to change.

My personal preference is to write constants as you see in the examples – all upper case with underscores replacing spaces. This is known as ‘screaming’ snake case. I do this to easily distinguish between regular variables as I’m reading the code. The Microsoft standard, though, is to use Pascal case, as in:

```
const int InchesInFoot = 12;  
const int OneDozen = 12;  
const int TaxRatePercent = 12;
```

Whichever approach you chose in your own code is up to you. If you are adding code to my codebase, however, I would expect you to follow my conventions and so would any other company you work for. The important thing is consistency so that readers of your code can ignore the format and concentrate on the functionality.

Variable scope

Variables have lifetimes. Once their lifetime expires, they are forgotten and the memory they consumed is returned to the pool of available memory through a process known as Garbage Collection (GC).

In C#, the nearest enclosing {} to its declaration statement determine the scope (or lifetime) of a variable. You can define a variable almost anywhere in your code if that definition is inside a pair of {}. As code executes from top to bottom within the {}, the variable stays alive. As soon as code execution hits the }, the variable is no longer accessible to your program and its memory is eventually returned to the available memory pool.

For now, the only {} we have represent the start and end of our event handler methods and the start and end of our MainWindow class. As we get more sophisticated, we will be created additional scopes within these two.

Text Input

To really do anything useful, we must collect some data to transform. Typically, this data comes from the application's user and is provided by the user as text via the keyboard. The user is expected to place the cursor into a text box and start typing – entering characters that make sense relative to what the application is asking. Once the user is done entering data into one or more text boxes, they are then expected to click a button to indicate that they are done.

In XAML, we use a `TextBox` for this purpose. We normally pair a `Label` control with a `TextBox` to give the user some clue as to what we are expecting them to type into the box. Something like this, for example:

```
<StackPanel>
  <Label Content="A label for the textbox" Margin="10,0" FontSize="12"></Label>
  <TextBox Margin="10,0" FontSize="25" Height="50"
    Padding="10,0" VerticalContentAlignment="Center"/>
</StackPanel>
```

Results in a `Label/TextBox` pair that looks like:

A label for the textbox

A screenshot of a Windows application window. The window has a title bar and a single content area. Inside the content area, there is a label with the text "A label for the textbox" in a small, dark font. Below the label is a text box with a light blue border. The text box contains the text "asdffdfadfsdfapppppgggggghhhh" in a large, dark font. The text is centered within the text box, and there is a vertical cursor at the end of the text.

Which has the look of a modern web application. Or, for a more traditional desktop application look, you can set the `StackPanel` orientation to horizontal. The reason for the different approaches between desktop and web applications is that on a desktop, users were traditionally not expecting to scroll vertically to access the whole form. It was supposed to be entirely visible. For a web page, however, the long dimension is down the page, so stacking label on top of text entry takes advantage of the narrow but tall area, with the limitation that scrolling is required. For a non-scrolling desktop application, the long dimension is across the page, so filling more of the horizontal space by putting labels beside text boxes was preferred. This is changing with Windows 10. For example, right click on the desktop and select the 'Display Settings' menu item. Notice that labels are on top and both left and right areas must be scrolled to access all the elements.

The example above is visually pleasing, but it is missing one important attribute. If we want to be able to access the contents of the text box, we need to assign a name to it so that we can use that name in our C# code, like:

```
<TextBox Name="inputText"
  Margin="10,0,10,0" FontSize="25" Height="50" Padding="10,0"
  VerticalContentAlignment="Center"/>
```

Once we have done that, then in our C# code, we can write:

```
string s = inputText.Text;
```


To access the text the user typed. This reads the contents of the textbox and copies it to a variable that we have defined. Typically, we would add this code to the event handler for a button click that is set to indicate the user wants us to do something with the data they have entered.

Converting text to numbers

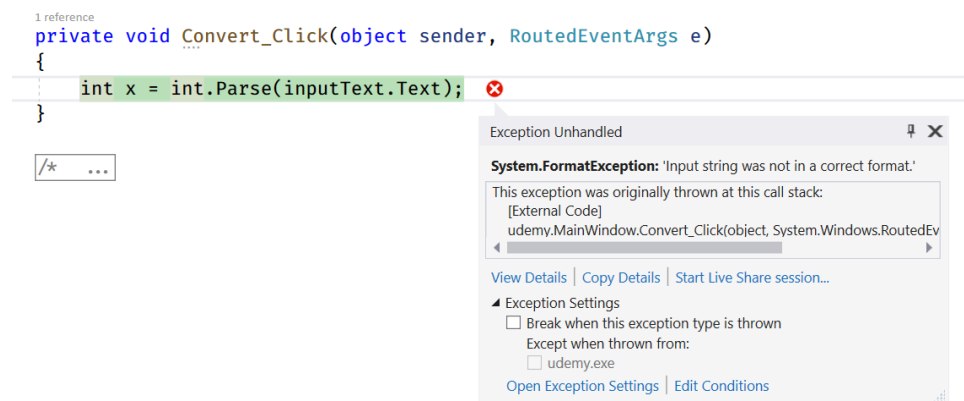
What the user types into a textbox will always be returned to us as a string, but sometimes we want numerical information. To extract numbers from the text, we need to convert or *parse* the text. Fortunately, the .NET Framework provides a mechanism for doing that. For instance, if you are interested in having the user enter an integer, then you can use this code:

```
int x = int.Parse(enteredText);
```

To convert the entered text into an integer that you can then operate on with math operations.

Try/Catch

If the user enters something that's not an integer, `int.Parse` will fail as it cannot return a valid integer. The way it expresses that failure is to throw an exception. In Visual Studio, an exception looks like this:



There are many reasons that an exception can be thrown by .NET when one of its built-in methods receives unexpected data. In this case, the exception is a Format Exception and it further specifies that 'Input string was not in a correct format'. This tells you that the user entered something that `int.Parse` did not recognize as an integer. Also, note that the title of the message box is "Exception Unhandled". An unhandled exception will cause the program to stop immediately with no feedback to the user – and that is something we do not want the user to experience.

Instead, we want to handle the exception and have the application continue. The way to handle exceptions is with the try/catch syntax:

```
try
{
    int x = int.Parse(inputText.Text);
}
catch
{
    MessageBox.Show("Please enter a valid integer");
}
```

With this code, if the Parse() method throws an exception, the next statement to be executed will be the first statement inside the catch scope. Now .NET knows what to do next after an exception – since we are telling it exactly what to do – and our application will continue to run. In this case, we are telling the user that we could not understand what they entered, and they can try again.

Since C# is a strongly typed language, its parsing is also strongly typed. That is, if your user enters 1.1 then int.Parse() will throw an exception. If you want to collect fractional numeric data, then you must use float.Parse(), double.Parse(), or decimal.Parse().

Four (actually 5) operations

There are 5 basic math operations built into all computer languages:

- Add, represented by + (shift = on a standard English keyboard)
- Subtract, represented by – (usually between 0 and =)
- Multiply, represented by * (shift 8)
- Divide, represented by / (bottom right on a standard English keyboard)
- Mod, represented by % (shift 5)

The first four operations are the normal operations you learned in grade school. The sequence of statements:

```
int a = 5;
int b = 2;
int c = a + b;
```

Results in c containing the value 7.

```
int c = a - b;
```

results in c containing the value 3.

```
int c = a * b;
```

results in c containing the value 10.

```
int c = a / b;
```

results in c containing 2. This one is perhaps a surprise since a normal division should yield a result of 2.5. However, the rules of C# are that an integer divided by another integer results in an integer. Since the integer part of 2.5 is 2, that is the answer. If you want to store the result 2.5 in a variable, then the code should instead look like:

```
float a = 5;
float b = 2;
float c = a / b;
```

A floating point value divided by another floating point value will result in a floating point value. This will also result in 2.5:

```
int a = 5;
float b = 2;
float c = a / b;
```

That is, only one of the two operands is required to be a floating point variable to have a floating point result.

The fifth operation, mod, is defined as the remainder after dividing two numbers. So this code:

```
int a = 5;
int b = 2;
int c = a % b;
```

results in the variable c containing the value 1. The result is because 5 divided by 2 is 2 with 1 remainder. As well, this

```
int a = 11;
int b = 7;
int c = a % b;
```

results in c containing 4, since 11 divided by 7 is 1 and 4 remainder.

This operation is commonly used when trying to determine if a number is odd or even. Any number % 2 will result in either 0 for even numbers or 1 for odd numbers.

You can also use it in combination with integer division to convert seconds into minutes and seconds, like this:

```
int totalSeconds = 113;
int minutes = totalSeconds / 60;
int seconds = totalSeconds % 60;
```

This will result in minutes containing 1 and seconds containing 53.

Math operations follow the BEDMAS (brackets, exponents, division, multiplication, addition, subtraction) rules, with mod at the same level as divide and multiply. For example:

- $5 + 11 \% 11 * 5$ equals 5 (% then * then +)
- $(5 + 11) \% 11 * 5$ equals 25 (() then % then *)
- $5 + (11 \% 11) * 5$ equals 5 (() then * then +)
- $5 + 11 \% (11 * 5)$ equals 16 (() then % then +)

The language, however, has no native exponent operator, so that does not factor into order of operations. To calculate an exponent, use a method from the Math library, like:

```
double y = Math.Pow(2, 3);
```

which is equivalent to 2^3 or 8. Check Microsoft's documentation for additional Math methods here (<https://docs.microsoft.com/en-us/dotnet/api/system.math>).

String concatenation

The + operator has a slightly different meaning when operating on strings. For example, this code:

```
string a = "1";
string b = "2";
string c = a + b;
```

results in the variable c containing the string "12". That is, the + operator concatenates the two strings. When dealing with mixed data types, the rules say this set of statements:

```
string a = "1";
int b = 2;
string c = a + b + 1;
```

will result in the variable c containing the string "121". That is because a string on either side of the + causes the other side to be converted to a string if it is not one already. And, since + are evaluated left to right, it first does the a + b operation and creates a string. Then the intermediate string + 1 causes the 1

to be converted to a string and concatenated. If you want the calculation $b + 1$ to be calculated first, then you must use brackets to force the order of operations to change, so that:

```
string a = "1";  
int b = 2;  
string c = a + (b + 1);
```

in this case, *c* now contains the string "13".

If you want spaces between words, then you must explicitly add them, like this:

```
string a = "1";  
int b = 2;  
string c = a + " " + b + " " + 1;
```

which results in *c* containing "1 2 1". Or, you can use string interpolation to create the same output, like so:

```
string a = "1";  
int b = 2;  
string c = $"{a} {b} 1";
```

Putting a *\$* in front of a string tells C# that you want to use string interpolation. What it does is take the contents of the variable, convert it to a string and place it in the final string as you lay it out. This makes for a shorter and more intuitive description of the final string output than the concatenation approach. It also makes it more explicit that each element gets converted to a string.

So, if you wanted to get the string "13", then you would write:

```
string c = $"{a} {b + 1}";
```

and the calculation of $b + 1$ is done first and then the result is converted to a string for inclusion in the string.

StringBuilder

Strings in C# are treated as immutable objects. That is, they cannot be changed. And yet, we can write things like:

```
string a = "1";  
a += "more string";
```

which seem to append more text to an existing string. Under the covers, however, what is happening is the system is creating a new string with the contents of the two original strings concatenated. Then it releases the reference to the original string and allows the garbage collector to clean it up. For small strings, this is not a problem. However, if we are working with very large strings then we will have two copies of the large string in memory at one time and we will need the system to spend time making copies, which will slow our application's performance.

To deal with this C# has a *StringBuilder* object type that eliminates these problems. The way to use it to perform the same actions as above is:

```
StringBuilder sb = new StringBuilder("1");  
sb.Append("more string");  
a = sb.ToString();
```

In this case every invocation of *Append()* really does append the new string to the existing without making a copy. Once completed, the *StringBuilder*'s contents must be converted to a string to be used. I

use this approach only if I am expecting to do a lot of appending or I expect the strings to be very large. Otherwise, concatenating strings directly is acceptable.

Formatting strings

At some point in our applications, we will be done with our calculations. Typically, we want to display the result to the user. For instance, when converting from Fahrenheit to Celsius the formula is:

$$\text{Celsius } (^{\circ}\text{C}) = (\text{Fahrenheit} - 32) / 1.8$$

If we use floating point calculations, 0°F converts to -17.7777778°C. However, users of our application likely do not care to see quite that many decimal places and will be happy with -17.8°C or even -18°C as an answer. To give them that, we need to format our output. Otherwise, we will display the full number of decimal places that a floating-point number holds to our users.

There are several ways to do this in C# code that all result in the same output:

```
string x;  
float f = -17.77777778f;  
x = f.ToString("n1");  
x = string.Format("{0:n1}", f);  
x = $"{f:n1}";
```

In each case, the variable x contains the string “-17.8”.

You can provide a parameter to the ToString() method specifying the format string. The string class has a Format() method where, in {} brackets, you identify the parameter by number followed by a format string. String interpolation allows for a format string after the colon. The format string – in this case ‘n1’ – means to display the number with one decimal place. N2 means two decimal places.

You can also use a format string like “#,##0.0” to indicate optional characters (#), required characters (0) and that you want the thousands separator (.). In addition to N, you can also use C for currency and P for percentages. For currency, it will automatically insert the currency symbol appropriate for your computer’s location and the right number of decimal places. For percentages, P will automatically multiply by 100 and add the % symbol so you can keep your internal numbers at the appropriate scale for calculations.

There are many other possible formatting strings, especially for dates. See the Microsoft documentation for the full list of formatting values:

- <https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-date-and-time-format-strings>
- <https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings>
- <https://docs.microsoft.com/en-us/dotnet/standard/base-types/custom-numeric-format-strings>

In addition to formatting in the C# code, you can also format text contents of Label controls in the XAML in this way:

```
<Label ContentStringFormat="N1"></Label>
```

You can use all the same format strings here as you can in the C# code.

Displaying columns of numbers/money

When displaying columns of numbers, especially money, it is important for comprehension that the decimal points be lined up. To do that, you should always show the correct number of decimal places (in North America, that is two) and align the text display to the right in the XAML using:

`HorizontalContentAlignment="Right"`

For example, it is much easier to understand the right column than the left.

1.59	\$1.59
1000	\$1,000.00

Currency symbols, right alignment, the same number of decimal points, and thousands separators each contribute to making it easier for the user to understand.

A final note on calculations. It is sometimes tempting when displaying individual line items on an invoice, for example, to calculate the total separately from the total of all subtotals. This will cause problems if fractions of a penny are involved. To illustrate this, consider:

Base cost: \$9.49	Tax in (@13%): \$10.7237	Rounded Amount: \$10.72
Base cost: \$10.49	Tax in (@13%): \$11.8537	Rounded Amount: \$11.85
Total pre tax: \$19.98	Tax in total: \$22.5774 or \$22.58	Rounded Total: \$22.57

If you calculate the tax for each item separately and round each result, the total is \$22.57. However, if you sum the amounts and then apply the tax, the total comes to \$22.58, or different by one penny. This may not seem like a big deal, but it is, and no knowledgeable user will accept the difference. As which is the correct amount, it depends. This is a business decision. Whatever the decision, though, you cannot mix the two approaches.

Exercise 3

A small dairy, Cherry on Top Creamery, makes ice cream, and they want to calculate the cost. The three main ingredients are cream (\$5.50/liter), sugar (\$1.50/kg), and eggs (\$0.25/each). Create an application that allows the user to enter the amount of each ingredient required for a batch of ice cream. The application must display the cost of each ingredient, the percentage of the total cost for each ingredient, and the total cost. Once you have completed the project, compare your work with Exercise 3.

Making decisions

If/else syntax

So far, we have let the user make choices by clicking one of several possible buttons that we have presented to them. Often, however, we need to make decisions in code and execute one or another set of statements based on some criteria. For example, instead of using two buttons to hide and show some element in our window, we might instead have a single button that allows the user to toggle between the two states.

The syntax for decision making is the *if* statement:

```
//if (<some code that evaluates to a boolean>
//{
//    <some statements to execute only if code in () above evaluated to true>
//}
```

A boolean data type has one of two values: true and false. What that means is there can be no ambiguity. Either the code block attached to the if will be executed or it will not.

Boolean logic

The simplest form of statement that will evaluate to a boolean is just a boolean, like this:

```
bool b = true;
if (b)
    MessageBox.Show("true");
```

It normally is not set up so directly. Usually, the state of the boolean is set by one part of the code and the test is done in some other part. If the scope of the boolean variable makes it accessible to both pieces of code, it can be a way to communicate a state.

Some controls and built in methods will return boolean values that can be used directly like this. For example, a TextBox control can be enabled or disabled. That is, the user may type text into the control. We can check the state of the control by

```
if (inputText.IsEnabled)
    MessageBox.Show("You are allowed to enter values");
```

The IsEnabled property of inputText is a boolean data type and so we can use it directly in the body of the if.

Comparison operators

Most of the time, however, we will have a variable that is not a boolean. In that case, we need to convert it into a boolean by using a comparison operator. The available comparison operators are:

- == (equality)
- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or equal to)
- != (not equal to)

It is important to note that the opposite of $>$ is $<=$, and opposite of $<$ is $>=$. Since neither > 0 nor < 0 include zero, we would be missing that value in both cases. However, > 0 and $<= 0$ are opposites because whatever part of the number line > 0 applies to, $<= 0$ applies to the remaining values exactly.

To use the comparison operators, the syntax looks like:

```
if (x < 0)
    MessageBox.Show("less than zero");
```

Instead of inverting the condition, it is also possible to invert the result using the `!` operator. Writing:

```
if (!(x >= 0))
    MessageBox.Show("less than zero");
```

means the same as the previous example, but with an extra step. Generally, simplify conditionals to make it less complex for the reader to understand the intent.

A warning when comparing float and double data types. Due to the possibility of rounding errors, comparing to an exact value is never a good idea. Instead, use a compound comparison to determine if two numbers are within an acceptable error term from each other, generally like this:

```
float f1 = 0.1f;
float f2 = 0.1000001f;
float err = 0.00001f;

if (Math.Abs(f1 - f2) < err)
    MessageBox.Show("consider them to be equal");
```

If the absolute value of the difference between the two numbers is less than an acceptable amount of error, then consider the two numbers to be equal. The actual precision will depend on the application. Of course, if you have two decimal data type values, you can use the `==` comparison, since decimal data type will not have rounding errors.

In addition to comparing integers and other numeric values, it is also possible to compare strings. For example:

```
string a = "1111";
if (a == "1111")
    MessageBox.Show("a contains 1111");
```

In this case, we are checking for equality. It is also possible to test for inequality by using the `!=` in place of `==`. However, if we want to determine if one string is larger or smaller than another, we cannot use the `>` and `<` operators. Instead, we need to use the `Compare` method of the string data type:

```
string s1 = "a"; //97
string s2 = "A"; //65

int r0 = string.Compare(s1, s2); //r0 is -1 'a' precedes 'A' in sort order
int r1 = string.Compare(s1, s2, false); //r1 is -1 'a' precedes 'A' in sort order
int r2 = string.Compare(s1, s2, StringComparison.CurrentCulture); //r2 = -1
int r3 = string.Compare(s1, s2, StringComparison.InvariantCulture); //r3 = -1
```



```
int r4 = string.Compare(s1, s2, true); //r4 is 0
int r5 = string.Compare(s1, s2, StringComparison.OrdinalIgnoreCase); //r5 = 0

int r6 = string.Compare(s1, s2, StringComparison.Ordinal); //r6 = 32 since 97-65=32
```

Compare returns 0 if the two strings are the same, a negative number if the first string comes first in the sort order, and a positive number if the first comes after in the sort order.

As you can see in the examples r0 through r3 above, all the options will tell you that 'a' comes before 'A'. However, depending on your system's settings, using the StringComparison.CurrentCulture option could, for some strings, provide a different value. That is because different languages can have different customary orders for their alphabet.

Examples r4 and r5 show case insensitive comparisons. In that case, of course, 'A' and 'a' are equal, so return zero values.

Finally, r6 shows the results of comparing the two strings using their 'ordinal' or Unicode values. In this case, the comparison returns a positive number because even though it is customary in English for lower case characters to precede upper case characters, they appear in the reverse order in the Unicode chart. See <https://unicode-table.com/en/> for the ordinal order of all Unicode characters.

Compound operators

At times, we are required to test multiple conditions. To accommodate that, we can use compound conditions, connected with:

- && (Conditional AND – both comparisons must be true for the test to be true)
- & (Logical AND – as above but can also work on integers)
- || (Conditional OR – either comparison must be true for the test to be true)
- | (Logical OR – as above but can also work on integers)
- ^ (Exclusive OR – two boolean values must be different for the test to be true; if used on integers, it operates on each bit of the integer individually)

The conditional operators require both sides to evaluate to a boolean value. If not, Visual Studio or the compiler will complain. However, be aware that the conditional operators take shortcuts if possible. The second part of the condition may not necessarily be evaluated if the first half guarantees that the whole will evaluate to either true or false. For instance, if the first half of an && test evaluates to false, the whole thing must evaluate to false, so there is no point in evaluating the second half of the condition. This also holds for the || operator. If its first half evaluates to true, then the whole must evaluate to true, and the second half will not be evaluated. When doing simple tests, this is inconsequential. However, if the second half is a method, the method may not be executed – and any side effects it may have produced will not occur.

The logical operators and XOR can be boolean values or integers. If integers, then bitwise arithmetic is applied – that is, each bit of the two integers is evaluated separately. For example, `7 | 8 == 15` is true, since 7 is 00000111b and 8 is 00001000b. Applying the Logical OR yields 00001111b or 15, since an occurrence of 1 in a position in either number results in a 1 in the result. Only if both numbers have a 0 in a position does the result end up 0. Unlike conditional operators, both sides of a logical operator are always evaluated. This is probably the most important distinction between the two types.

The syntax for using these operators is like:

```
if (x >= 0 && x <= 7)
```

which will be true if x is 1, 2, 3, 4, 5, or 6, and false everywhere else.

Where the non-evaluation of the second part of the conditional AND comes in handy is when you only want to execute the second half if the first half is true. For example, in this scenario we want to test if the control is visible, but only if `getControl()` returned one.

```
Control c = getControl();  
  
if (c != null && c.Visibility == Visibility.Visible)  
    MessageBox.Show("control is visible");
```

If c is null, then the second half of the if is not executed and the code will not throw an exception.

Over specifying conditions

A common problem among new developers is a desire to over specify tests. It is important not to do this as it generally comes back to cause extra work later. If the specification says that you only need to consider green dragons in your code, you might want to write code like:

```
if (isDragon(d) && d.Color == Colors.Green)
```

but if the statements associated with the if do not have anything to do with color, that second half is meaningless. More importantly, if tomorrow the specification changes so that golden dragons are also to be processed, then this code needs to change. However, it would not have needed to be updated if the conditions were kept to only those that were required.

Additional conditions can also lead to bugs. This is especially true in if/else if chains. For example, if we are trying to convert number grades into letter grades, then it is common to find code like this:

```
string letterGrade = "";  
if (numberGrade > 85 && numberGrade <= 100)  
    letterGrade = "A";  
else if (numberGrade > 75 && numberGrade < 85)  
    letterGrade = "B";  
else if (numberGrade > 65 && numberGrade < 75)  
    letterGrade = "C";  
else if (numberGrade < 65)  
    letterGrade = "F";
```

This specifies too much. In fact, the second half of all the conditionals can be removed. But worse, this code contains errors. Pause and see if you can spot them.

The error is that anyone who got exactly 85, 75, or 65 will get NO letter grade. That would also be true of anyone who received more than 100 – perhaps due to some bonus marks.

A better approach is to write this:

```

if (numberGrade > 85)
    letterGrade = "A";
else if (numberGrade > 75)
    letterGrade = "B";
else if (numberGrade > 65)
    letterGrade = "C";
else
    letterGrade = "F";

```

This produces the same results in the normal cases but does not suffer from the missing values error AND is easier to read because there is less code.

Remember that once one branch of an if tree is taken, the remaining branches are ignored, so if the letter grade A is assigned, it will not be changed by the remaining code.

The if/else if tree could also be rewritten this way to again get the same result:

```

if (numberGrade <= 65)
    letterGrade = "F";
else if (numberGrade <= 75)
    letterGrade = "C";
else if (numberGrade <= 85)
    letterGrade = "B";
else
    letterGrade = "A";

```

Notice that this time the code uses the <= conditional and starts with the lowest value and work its way up. The previous version uses the > conditional and starts with the largest value. This is no coincidence, but a rule to ensure the branches are taken appropriately.

Ternary operator

There is a special class of if statement where we want to assign a different value to the same variable depending on which branch is taken. For example,

```

If (x >= 0)
    s = "Positive";
else
    s = "Negative";

```

Because this is so frequent, there is shorthand for writing it, called the Ternary operator. It looks like this:

```
s = x >= 0 ? "Positive" : "Negative";
```

These both do the same work, but the ternary operator uses less text to express it. So, of course, programmers favor this terse approach. It is, however, possible to overdo the ternary operator. Two levels deep is the maximum before it becomes more difficult to read. Stick to if/else if for more options. A two level test would look like:

```
string s = x > 0 ?
    "Positive" :
    x < 0 ? "Negative" : "Zero";
```

which would be written in if/else if form as:

```
string s;
if (x > 0) s = "Positive";
else if (x < 0) s = "Negative";
else s = "Zero";
```

Indentation will have a significant effect on readability as well. Although the ternary version could have been written on one line, this three line approach makes it clearer which is the true and which is the false condition.

Boolean Assignment

Another special case is if we need to assign a true or false value to a boolean depending on the result of the comparison, like so:

```
If (x > 0)
    b = true;
else
    b = false;
```

Remember that the comparison itself creates a boolean value, so this can be written more simply as:

```
b = x > 0;
```

Using the first approach (or even a ternary version) is a sure sign of a new programmer. Understand what is happening in your code and take the shortest path to accomplishing what you need.

Switch

There is a different approach to making decisions that is more suitable for a set of discrete values. The older switch statement syntax looks like:

```
switch(letterGrade)
{
    case "A":
        numberGrade = 85;
        break;
    case "B":
        numberGrade = 75;
        break;
    case "C":
        numberGrade = 65;
        break;
    default:
        numberGrade = 45;
        break;
}
```

The newer syntax, referred to as a “switch expression” looks like:

```
var numberGrade = letterGrade switch
{
    "A" => 85,
    "B" => 75,
    "C" => 65,
    _ => 45,
};
```

It is not always possible to convert a switch statement into a switch expression. Use switch expressions when each option sets the same variable.

The datatype of the value and the constant can be any simple data type, but they must match. This reduces the amount of text required to write the equivalent if/else if chain in those cases where the comparison would have been equality (==). This makes for a more readable code block, especially when there could be many discrete values. The switch expression reduces the extra text even more.

Enum

Enums are a C# language construct that have the same functionality as constants, but are usually associated with a group of values where we really do not care about the actual integer associated with each element in the group. For example, in a chess game, we might want to distinguish between the various pieces. We could use constants, like:

```
const int ROOK = 0;
const int KNIGHT = 1;
const int BISHOP = 2;
const int QUEEN = 3;
const int KING = 4;
const int PAWN = 5;
const int MAX = 6;
```

But we really do not care if PAWN is 6 or 1, just as long as it is not the same value as any other piece. As well, it would be better if IntelliSense could prompt us when we need to remember what we called something. To deal with these problems, we could use an enum, like so:

```
enum Piece
{
    ROOK, //is by default assigned the value zero
    KNIGHT, //incremented by one from the previous value
    BISHOP,
    QUEEN,
    KING,
    PAWN,
    MAX
}
```

The language assigns a unique value to each item listed in the enum. We can choose the value that is assigned, but generally we do not. Note the MAX enum value. Since each item is assigned a monotonically increasing integer value starting at zero, MAX will be assigned a value equal to the number of distinct pieces. To get at the number, it is possible to cast the value, so for example:

```
int x = (int) Piece.MAX; //equivalent to x = 6
```

can be used to convert the enum's value to be used as an integer. The main purpose, however, is to group like values together to make them easier to use consistently. You have already experienced enums when you used the `Visibility.Visible` construct to set a control's visibility. Although you can use enums in if statements, they are discrete values, so ideally you would use them in a switch, like so:

```
int c = p switch
{
    Piece.BISHOP => (int) p,
    Piece.KING   => (int) p,
    Piece.KNIGHT => (int) p,
    Piece.PAWN   => (int) p,
    Piece.QUEEN  => (int) p,
    Piece.ROOK   => (int) p,
    _            => 0
};
```

Exercise 4

Canada has coins for in the following denominations: 5 cents, 10 cents, 25 cents, 50 cents, 100 cents, and 200 cents. Create an application that allows the user to enter a money amount between 0 and 2000 cents in 5 cent increments into a text box. The program should display the minimum number of coins required to reach that amount. If the number is outside the range or if it is not a multiple of 5, the program should display an error message. For example, entering 80 should result in 1 - 50, 1 - 25, and 1 - 5.

Use only if/else and switch statements in your code. HINT: the mod (%) operator will be helpful here. Once you have completed, compare your work to the project Exercise 4.

More controls

Checkboxes

When collecting information from our application's user, we want to constrain their selections to make it clearer what the choices are and to make it easier to write the code. We could, for example, collect the answer to a yes or no question with a text box control, but the range of possible answers is enormous:

Yes, Y, y, yes, agree, I agree, approve, go ahead, please, OK, ok

Are just some of the English words that can be used to indicate an affirmative response in an unconstrained text box. Rather than making the user, and the programmer, guess the range of possible responses, it is much better to provide a control where the response is a boolean value.

In WPF, we represent a binary choice we want the user to make with a checkbox. This checkbox usually has some text beside it that indicates the choice the user is to make. Normally, the phrasing is such that the positive/true/checked state is expressed in the words. For example, "Use Warp Drive" would indicate that if checked, the system will use its warp drive. Be careful of your phrasing, as it is possible to create confusion. For instance, if we have a checkbox that looks like:

☐ Ask Advertiser Not to Track

What does it mean to check? To uncheck? The problem is that there are actually three possible states: advertiser does not track, advertiser tracks, and advertiser must ask before being allowed to track. A checkbox is effectively merging two of these three options into one. In the case of iOS, unchecked means the advertiser cannot track and checked means the advertiser must ask, but then can track with a positive response. The problem is the wording does not make that clear.

Checkboxes are represented in the XAML by using the <CheckBox> tag:

```
<CheckBox Content="Ask Advertiser Not to Track"></CheckBox>
```

One of the issues with a visually pleasing checkbox is if you wish to adjust the font size of the label text. If you increase the font size, the checkbox itself does not get bigger, so you end up with something like:

☐ Ask Advertiser Not to Track

A better visual outcome happens if you use Viewbox instead, like this:

```
<Viewbox>
  <CheckBox Name="AskAdvertiser" Content="Ask Advertiser Not to Track"></CheckBox>
</Viewbox>
```

Because here the text and the checkbox graphic resize together.

To use this control in C#, you can check the state of the checkbox with the `IsChecked` property. Note that the `IsChecked` property is not a `bool`, but instead a `bool?` data type. This is a shorthand for the nullable set of data types:

```
Nullable<bool> isChecked = AskAdvertiser.IsChecked;
```

Is equivalent to:

```
bool? isChecked = AskAdvertiser.IsChecked;
```

In either case, the 'nullable' part indicates that there is one more state possible for the variable, and that is null. This allows us to represent SQL database records in objects more accurately – since SQL allows fields to be null as well as the usual set of values. For instance, a boolean in an SQL database may have one of 3 possible values: true, false, and null. It is possible to apply nullable to any simple data type, except for the string type – that is because strings can be null on their own without this extra indication.

Since the condition of an if statement requires that it evaluate to a `bool`, `IsChecked` is not an appropriate condition. Instead, we need to convert it to a boolean value by writing something like this:

```
if (AskAdvertiser.IsChecked == true)
```

This works because the equality comparator checks to see if the two sides are the same. If they are, that is a boolean true. If not, that results in false. So, `false == true` and `null == true` both are false. `True == true` is, of course, true.

Another bit of C# syntax that we can use is the `??` operator. This is called the null-coalescing operator and provides an alternate value for a null value. For instance, to solve the previous issue, we could also write:

```
if (AskAdvertiser.IsChecked ?? false)
```

This would have the same effect as the previous solution, but instead of converting by doing an equality comparison against true, we instead use the `??` to convert a possible null value to false. In other words, if `IsChecked` is false, the `??` does nothing. If `IsChecked` is true, the `??` does nothing. However, if `IsChecked` is null, then the `??` causes the null to be replaced with the value on the other side of the `??` – in this case 'false'.

This null coalescing operator can also be used on all other nullable types, so it is possible to write:

```
int? q = null;  
int s = q ?? 7;
```

which, in this case, would result in the variable 's' containing 7 once this code was executed.

Radio Buttons

Many times, the set of choices we want the user to pick from is three to seven. In this case, we usually represent this set of choices by using radio buttons. As with checkboxes, use `Viewbox` to resize the

visuals. Note, however, that a Viewbox can only contain one element, so you must enclose the set of radio buttons in a Grid or StackPanel, like this:

```
<Viewbox>
  <StackPanel>
    <RadioButton Content="No tracking allowed"></RadioButton>
    <RadioButton Content="Advertiser must ask before tracking"></RadioButton>
    <RadioButton Content="Advertiser can always track"></RadioButton>
  </StackPanel>
</Viewbox>
```

for it to look like:

- ☐ No tracking allowed
- ☐ Advertiser must ask before tracking
- ☐ Advertiser can always track

Radio buttons operate in a group, where only one at a time can be selected from that group. If you wish to have multiple groups, then they must be inside different parent elements. In the example above, adding a second StackPanel with its own set of RadioButton controls creates a second group. Another option is to explicitly give each RadioButton a group name using the `GroupName` attribute. That is less common, as typically you will want groups of radio buttons to be visually separated and that's most easily accomplished by having separate enclosing elements.

In C#, you access the state of a radio button with the `IsChecked` property – just like checkboxes. And, just like the `CheckBox` control, the `IsChecked` property is a `bool?` data type. You will want to name each of your radio buttons, because you will need to check each `IsChecked` property to find the one that the user selected. There is no master element you can interrogate to discover that information.

ListBox

If the number of options for a user to pick from exceeds 7, or the number is variable, or we need to provide the user with many options and we are trying to save some space in the layout, then use a `ComboBox` or `ListBox`. A `ComboBox` is effectively an occasionally visible `ListBox` plus an always visible `TextBox`. From a programming point of view, however, they are essentially the same. For that reason, we will focus on the `ListBox` in this discussion. Just know that, for the most part, what we say here is transferable to the `ComboBox`.

`ListBox` controls show a set of options from which one (or more) can be selected. In their simplest form, they can be populated directly in the XAML:

```
<ListBox Name="SelectList">
  <ListBoxItem>Item 1</ListBoxItem>
  <ListBoxItem>Item 2</ListBoxItem>
  <ListBoxItem>Item 3</ListBoxItem>
</ListBox>
```

They can also be populated via code by calling the `Add` method of the `Items` property:

```
SelectList.Items.Add("Item 4");
```

You may want to react immediately when the user selects an item from the list. This can be accomplished by handling the `SelectionChanged` event in just the same way you handle the `Button` control's `Click` event.

Mostly, though, you will wait until the user is finished making all their selections and then clicks a 'Save' `Button` control to indicate that they are finished entering information. In that case, you will want to see which of the items was selected. The most convenient way is via:

```
int selection = SelectList.SelectedIndex;
```

This gives you the index of the item selected. Use a constant or cast the selection integer to an enum to translate the value into something your code can use.

One thing you must be aware of in your code. If the user has not made any selection, the value returned by `SelectedIndex` will be `-1`. Your code needs to handle that situation -- probably with an `if` statement to validate selection.

Style

Up until now, we have modified the style of each XAML control individually. As the number of controls on the page increases, this starts to become repetitive. XAML provides a special tag -- `Window.Resources` -- where you can place styles that you wish to reuse. For example:

```
<Window.Resources>
  <Style TargetType="TextBlock">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="14"/>
  </Style>
  <Style x:Key="customBorder" TargetType="Border">
    <Setter Property="BorderThickness" Value="3" />
    <Setter Property="BorderBrush" Value="Black" />
  </Style>
</Window.Resources>
<Grid Margin="5">
  <Border Style="{StaticResource customBorder}">
    <TextBlock>the rain in spain</TextBlock>
  </Border>
</Grid>
```

This defines two styles: one unnamed `Style` targeting the `TextBlock` control and a named `Style` targeting the `Border` control. The unnamed `Style` will apply to all `TextBlock` controls in the window. The named `Style` will apply only to those `Border` controls that explicitly specify a named `Style` attribute. Use one or the other approach depending on whether you have a single style that will apply to every control of a specific type or whether you have several styles to select from.

In addition to `Window` level resources, you can also specify control scoped resources with something like `Grid.Resources`. Resources defined in this way are valid only within that control and is something you may want to do in a very complex layout. For most purposes, `Window` level resources are sufficient. If, however, you have many windows in your application, you may want to share resources across windows. You can do this by defining your resources inside the `Application.Resources` tag:

```
<Application.Resources>
</Application.Resources>
```

Found in the app.xaml file created for you when the project files were instantiated.

XAML also provides the ability to create animations. Animations can be useful to subtly notify the user that some action has taken place, giving useful feedback that the user's input has been consumed and reacted to. For instance, if a new row is added to a ListBox control, showing an animation will draw the user's attention to it. If it were to just suddenly appear, the user might not notice at all. In the following example, any Button control on the page will animate by expanding and contracting when it is clicked.

```
<Window.Resources>
  <Style TargetType="{x:Type Button}">
    <Style.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation AutoReverse="True"
                Storyboard.TargetProperty="RenderTransform.(ScaleTransform.ScaleX)"
                From="1.0" To="1.1" Duration="0:0:0.1" />
              <DoubleAnimation AutoReverse="True"
                Storyboard.TargetProperty="RenderTransform.(ScaleTransform.ScaleY)"
                From="1.0" To="1.1" Duration="0:0:0.1"/>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
<StackPanel Margin="10">
  <Button Content="Do it" Width="100" Height="25">
    <Button.RenderTransform>
      <ScaleTransform CenterX="50" CenterY="12.5" />
    </Button.RenderTransform>
  </Button>
</StackPanel>
```

As before, we are setting a Style with a Button control as the target. However, since we want to react to events, we need to define an EventTrigger and we will say that the event we want to react to is the Button.Click event. In the EventTrigger.Actions, we then want to define a Storyboard – this is the name given in XAML for animation descriptions. The DoubleAnimation tag defines animations applied to values expressed as the Double data type (it does not double the animation). To expand, we ask the animation to start at 1.0 times scale in both x and y and expand to 1.1 time scale. By specifying the AutoReverse property, the animation will reverse itself and return to the starting state. The Duration property specifies that this whole thing should take 0.1 seconds. In the button, we specify a RenderTransform property where we specify the center of the scale change. By placing it in the center of the button, it expands equally in all directions around the center.

Exercise 5

Create an application to collect survey data. Use a radio button group to collect gender (male/female/other). If they pick other, enable a ComboBox from which they can choose one of (from

Facebook): Agender, Androgyne, Androgynous, Bi-gender, Cis, Cis-gender, Cis Female, Cis Male, Cis Man, Cis Woman. Use a ListBox to allow the user to pick a favorite musical genre (Rock, Country, Classical, etc.). Use a set of checkboxes to allow the user to pick their listening modes (Streaming services, Satellite Radio, Radio, CD, LP, etc.). When the user presses a button, display a summary of their choices in a TextBlock. Once completed, compare your work to Exercise 5.

Class Basics

At this point, it is necessary to introduce the concept of 'Class'. You can think of a class as an architectural blueprint. Just like a house plan describes a house, but is not itself a house, A Class is, usually, contained in a file with code describing the properties and behaviors of the object being described. For example, we might need, in our application, to represent an egg. We might write something like:

```
namespace App
{
    public enum EggSize
    {
        PeeWee,
        Small,
        Medium,
        Large,
        ExtraLarge
    }
    public class Egg
    {
        public EggSize Size { get; set; }
        public decimal Price { get; set; }
        public bool IsHardBoiled { get; set; }
    }
}
```

Our Egg class is defined to have three properties, Size, Price, and whether this egg is hard boiled or not. A couple of things to notice. Classes are by convention named after a noun (a thing) and they are first letter capitalized. All the public properties are also named after things (nouns) and they are first letter capitalized.

Notice the {get; set;}. Including this is what distinguishes a property from a variable. It may appear that they are interchangeable, but properties have some additional behaviors that make them the correct way to define attributes in a class. The best way to illustrate this is to write the Egg class out as it would have been before the {get; set;} shortcut syntax was introduced:

```
public class Egg
{
    private EggSize size;
    public EggSize Size { get { return size; } set { size = value; } }

    private decimal price;
    public decimal Price { get { return price; } set { price = value; } }

    private bool isHardBoiled;
    public bool IsHardBoiled { get { return isHardBoiled; } set { isHardBoiled = value; } }
}
```

This gives us the ability to write any arbitrary code and do any kind of calculation inside the get and set. This provides flexibility that a variable cannot match, but still offers the syntax of a variable to users of the class. The get is invoked whenever you get the contents of the property. The set is invoked whenever you update the property.

It is also possible to leave off the set; or to define the set; as private. A typical case for a property without a set is one that does some calculation based on the other properties, like:

```
public string Description { get { return $"{Size} | {Price} | {IsHardBoiled}"; } }
```

Here we create a description that is based on all the other properties of the class, and not any external values. If the user of this class were to attempt to set the Description, they would receive a compile time error.

You might use a private set if a private calculation is involved in deciding on a property's value. For instance, we could rearrange the Description calculation like this:

```
public class Egg
{
    private EggSize size;
    public EggSize Size
    {
        get { return size; }
        set { size = value; Description = $"{Size} | {Price} | {IsHardBoiled}"; }
    }

    private decimal price;
    public decimal Price
    {
        get { return price; }
        set { price = value; Description = $"{Size} | {Price} | {IsHardBoiled}"; }
    }

    private bool isHardBoiled;
    public bool IsHardBoiled
    {
        get { return isHardBoiled; }
        set { isHardBoiled = value; Description = $"{Size} | {Price} | {IsHardBoiled}"; }
    }

    public string Description { get; private set; } }
}
```

Now, we recalculate Description whenever any of its parts change instead of waiting until a user of this class requests it. Which is the better approach depends on how often the other properties change compared to how often the Description is needed. If the properties do not change often but Description is often used, then calculating on change and storing the result is better. If, however, the properties change quickly, but the description is only rarely needed, then calculating it repeatedly is a waste of resources.

By defining this Egg class, we have described the attributes that we find important about eggs within our application. However, we have not yet created any actual eggs. To create an actual egg object in our application, we would need to write:

```
Egg egg = new Egg();
```

The 'new' keyword asks the system to allocate space for all the properties that we have defined for an egg. To access the properties of egg, we can write:

```

if (egg.IsHardBoiled)
{
    egg.Price = 0.59m;
    MessageBox.Show($"{egg.Price} {egg.Size}");
}

```

If we want more eggs, then we simply execute that line again, assigning the resulting object to a different variable.

Of course, you have been using a class all along. When Visual Studio built a basic application for you, it created a `MainWindow` class for your main window. It also automatically generates a `Main()` method, where it creates a new instance of the `MainWindow` class.

Notice that the `MainWindow` class inherits behavior from the built-in `Window` class. That's how we get all the 'window' behaviors for free. We tell it to inherit behaviors and properties of a different class by using this syntax:

```

public class MainWindow : Window

```

Finally, when we specify events to handle in XAML, it creates a place to put that code in the `MainWindow` class. These places are known as methods. We could also add methods to our `Egg` class, like so:

```

public class Egg
{
    public EggSize Size { get; set; }
    public decimal Price { get; set; }
    public bool IsHardBoiled { get; set; }

    public void Boil()
    {
        IsHardBoiled = true;
    }
}

```

This follows approximately the same pattern as the auto generated methods. Again, by convention, the method is capitalized since it is public. More importantly, the name is an action word (a verb). Methods, for the most part, act on properties to effect some transformation.

Methods

A method is a named group of statements that we want to repeatedly use as a unit of work. The name is important. It must describe what the method does with as few words as possible. However, those words must convey an action.

A method should take a small set of (or no) inputs and cause some change in some data or provide some result. If no change occurs to the data or visual appearance of your application, and the method provides no information, then there was no purpose to the method and it should not be used and should be removed from your code base.

DO NOT LEAVE unused code in your project. Remove it aggressively.

You have been using methods created by others throughout this book. Parse(), for example, is a method. The Show() of MessageBox.Show() is a method. Finally, you have been populating the event handler methods with your own code right from the beginning.

You can, of course, create your own methods to create your own encapsulated behaviors. The simplest method syntax looks like (note, not public, so convention is to use a lower case name):

```
void boil()
{
}
```

Where the () describes the parameters accepted by the method. In this case, the method will not expect any parameters. As always, the {} indicate the scope of any variables declared inside the method.

As you start writing your code, you may give the method a name thinking it will have a certain behavior. But, as you continue development, you may find that it is convenient for the method to change its behavior to do something more or perhaps even something completely different. If that happens, YOU MUST RENAME the method. When someone else is scanning your code, a poorly named method wrongly identifies the encapsulated behavior and that causes confusion. A very common mistake is to start with a method like:

```
bool checkSomething()
{
    return false;
}
```

And then later change it to this:

```
bool checkSomething()
{
    if (false)
        doSomethingElse();
    return false;
}
```

Without changing its name. Now, this method has multiple duties. It is checking, but it also makes some other change – something not captured in the name and thus missed in a quick scan of the code. Better to call it:

```
bool updateSomethingElse()
```

with the return value letting us know whether the update occurred or not.

Parameters

Sometimes a method can do its work based on variables already in its scope, but many times you want to provide some values so the method can perform variations of a task. For example, the MessageBox.Show() method takes a string parameter for the message to show. If it did not, we would have to have as many show methods as there are possible messages to display – something impossible to do in practice.

Parameters can be any simple data type (int, string, etc.) or with any built-in class (Window, Button, ListBox, etc.), or any class you create. Defining a method with parameters looks like:

```
void paramMethod(int p1, string p2, Window p3)
{
}
```

And using that method looks like:

```
int x = 1;
paramMethod(x, "hello", sender);
```

You can provide parameters as variables or as hard coded values directly – as you have already done for `MessageBox.Show()`.

Parameters have a scope, just like any other variables you declare in your code. Consider this method:

```
void changeMethod(int x)
{
    x = 2;
}
```

And call the method with this code:

```
int x = 1;
changeMethod(x);
if (x == 1)
    MessageBox.Show("x is 1"); //this will be executed
```

Notice that the changes to the variable made in one scope do NOT change the variable in the other scope, even if it has the same name. This is what is known as passing parameters by value. What is going on is that a copy of the value contained in `x` is provided to `changeMethod` and assigned the name `x`. Even though it is the same name, it is NOT the same location in memory. It is only a copy. This code changes the copy's value to 2. Since we've only changed the copy, once `changeMethod()` is done, the variables within the `changeMethod` scope are destroyed, and control is returned to the original scope, we find that the value of `x` in this outer scope is unchanged.

Ref

If we do want to allow a method to change the value of a parameter in the outer scope, then we can use the `ref` keyword to accomplish this.

```
void changeMethod(ref int y)
{
    y++;
}
```

Now if we call the method:

```
int x = 1;
changeMethod(ref x);
if (x == 1)
    MessageBox.Show("x is 1"); //this will NOT be executed
```

This time, we pass in a reference to the memory location containing the value of x, not a copy. Even though we call it something different in the called method, on return, the contents of the memory that we have labeled x have changed.

We only need to use the 'ref' keyword for simple data types. Any other data types, such as objects created from either built-in classes or classes you have created, are always passed by reference. The reasons for this are twofold. First, objects are potentially very large and making a copy could put an unreasonable strain on the memory available, and increase the time required to create and destroy the copy of the object.

Second, we normally want to manipulate the original object to make the change permanent. Imagine passing a TextBox object to a method and having that method change the Text property, only to have the change thrown away once the variable went out of scope because the change was made to a copy, and not the original. It is very rare that we only want to manipulate the copy, but in those cases, you would explicitly make the copy before passing it around.

Out

We sometimes want to have a method do some calculation and provide us with an answer in the way that the ref keyword will modify the variable. However, the method does not use the original value of the variable for anything. In that case, rather than using ref, we can use the out keyword – like TryParse() does. TryParse() does not use the input variable's value at all, so out is a better match than ref.

For example, if we define a method with an out type parameter:

```
void changeMethod(out int y)
{
    y=2;
}
```

Then if we call the method:

```
int x = 1;
changeMethod(out x);
if (x == 1)
    MessageBox.Show("x is 1"); //this will NOT be executed
```

Since we do not actually use the value of x until after changeMethod is called, we can delay defining the variable, and write this code defining the variable in the call. Note the use of var, the compiler knows the data type of y based on the signature of the method.

```
changeMethod(out var y);
MessageBox.Show($"y is {y}"); //y is 2
```

Again, we pass in a reference to the memory location containing the value of x, not a copy. But, because we used out, the method ignores the value stored in that memory to start and expects that the method will set it. In fact, if the method forgets to set it, Visual Studio will complain, and the code will not run.

Default value

One more thing we can do to parameters is to assign them a default value. You do this by changing the method signature to look like:

```
void defaultValueMethod(int x = 0)
```

which says that if the user provides no value for the first parameter, then assign the default value of zero. It then becomes possible to call the method in one of these two ways:

```
defaultValueMethod();  
defaultValueMethod(1);
```

There limitation is that parameters with default values must appear at the end of parameter list. Otherwise, the compiler may not be able to figure out which parameters you are supplying, and which are to be defaulted. In older versions of C#, this was not available and could be simulated by overloading methods. That is, you could create multiple methods with the same name but varying the parameter lists. So, it is possible to also write:

```
void overloadedMethod(int x)  
{  
}  
  
void overloadedMethod()  
{  
    overloadedMethod(0);  
}
```

To get the same effect. The default value approach was added to eliminate the need for writing this code. However, there are still legitimate uses for overloading. For example, the methods of the built-in `System.Convert` class each take various data types and convert them to the data type specified by the method name. For instance, `Convert.ToInt16()` can be passed one of 18 different data types and will attempt to produce an `Int16` value on return.

Methods as parameters

In more advanced scenarios, it may be useful to have a common method that implements the basic outline of some algorithm, but then provide it with a custom method for the various specific use cases. For this reason, we can also pass methods as parameters. For example, this set of methods demonstrates a simple use case:

```
int calc(int a, int b, Func<int, int, int> f)  
{  
    return f(a, b);  
}  
int add(int a, int b)  
{  
    return a + b;  
}  
int div(int a, int b)  
{  
    return a / b;  
}  
void useCalc()  
{  
    int addResult = calc(1, 2, add); //returns 3  
    int divResult = calc(2, 1, div); //returns 2  
}
```

Here the method `calc()` takes two integers and a `Func` as parameters and returns an `int`. Internally, `calc()` simply calls the provided method with the other two parameters and returns its result. The types defined inside the `<>` of the `Func` describe the data type and number of the parameters, followed by one last data type that describes the required return type. Notice that both `add()` and `div()` follow this pattern. If they did not, then a compile time error would be raised. Finally, `useCalc()` shows them in action. Notice that when passing a method as a parameter, it does not include the `()`. If you did write `add()`, for example, that would fail to compile as it would be attempting to invoke the `add` method with no parameters.

In addition to `Func<>` to describe a method with a single return value, you can use `Action<>` in the same way to describe a method without a return value.

Single Return value

The keyword `'void'` of the previous example methods is a required part of the method signature and indicates the value the method will return. In this case, the method will not return any value (or `void`). It is possible to replace `'void'` with any simple data type (`int`, `string`, etc.) or with any built-in class (`Window`, `Button`, `ListBox`, etc.), or any class you create.

When calling a method that has a `void` return value, you simply write

```
otherMethod();
```

If, for example, we would like our method `intMethod()` to return an `int` value, we need to replace `void` with `int`, and it would look like:

```
int intMethod()
{
    return 0;
}
```

Notice the new `'return'` keyword, followed by a value. This is required for a method that returns a value. Of course, the value's type must match the type in the method signature.

You would call this method in this way:

```
int x = intMethod();
```

Where the result of the method's calculations are saved to the current scope in the variable `x`.

It is possible to ignore the returned value implicitly by

```
intMethod();
```

Or explicitly by

```
_ = intMethod();
```

The underscore character, which we've seen in the default case of the `switch` is used to acknowledge that a value is returned, but that the code has no need to remember the value. Depending on the coding

style chosen by the team, one or the other approach is used. Most of the time, however, the first option is what is written.

It is possible to have multiple return statements in a single method. Generally, these are inside if statements. Otherwise, any code after the return will never be executed. You can use this style if you need to do some validation before your main block of code, like:

```
int addOne(int q)
{
    if (q < 0)
        return 0;
    if (q > 1000)
        return 1000;
    return q++;
}
```

Here we are validating that the parameter passed in are in the correct range before we continue to the main logic of the method. Another way to write this is:

```
int addOne(int q)
{
    int retVal;

    if (q > 0)
    {
        if (q < 1000)
            retVal = q++;
        else
            retVal = 1000;
    }
    else
        retVal = 0;

    return retVal;
}
```

The advantage of the multi-return approach is that it does not require as much nesting of statements. On the other hand, having multiple places where a method could exit makes tracing the logic more difficult. When taking this approach, limit yourself to early exit on validation and have all the early exits near the start of the method.

The single return approach is more nested, but it does make debugging easier since you always know where the logic will end up before exiting. To keep the nesting to a minimum, encapsulate your validation logic in their own methods and use compound if conditions when testing them.

Which approach you chose in your code is mostly a personal preference. Do look, however, at existing code in the project you are adding to and follow along with whichever approach the other developers in the project are using.

Tuple returns

The Tuple class has been in C# since version 6, but the syntax made it difficult to use. Improvements in later versions have made it a convenient way to return multiple values. To use the tuple as a return value, you would write a method that looks like this:

```

(int, string, bool) tupleMethod()
{
    int x = 7;
    bool z = false;
    return (x, "test", z);
}

```

And it is called this way:

```
(int x, string y, bool z) = tupleMethod();
```

This approach can be used as a direct replacement for out type parameters. Although it is possible to create a tuple with up to eight values of various types, you will want to keep this to under five. If you must return more values, create a class, and return that.

Using the Lambda operator

Another way of writing methods is to use the => syntax. Here are two ways to write the same method:

```

bool method(bool x) => !x;

bool method(bool x)
{
    return !x;
}

```

For one-line methods, the Lambda operator (=>) replaces the {} and return – making for a more compact representation. It also makes for a more convenient approach to writing anonymous methods. We will look at that in more detail in later sections.

MVVM

MVVM (or M V VM) is an acronym that stands for Model View ViewModel and is a way of organizing applications that separates the application into these three components connected with a loosely coupled arrangement. The intention is to create a more flexible and easily modified application structure. It also has the benefit of eliminating code that we would otherwise write.

To this point, we have been writing tightly coupled code. That is, we mix the business logic and the presentation in a way that changing either requires that both types of code are affected. For example, a very simple tightly coupled implementation of summing two numbers might look like:

```
private void Calc_OnClick(object sender, RoutedEventArgs e)
{
    var sum = 0;

    if (int.TryParse(value1.Text, out var v1) &&
        int.TryParse(value2.Text, out var v2))
        sum = v1 + v2;

    result.Text = sum.ToString("C");
}
```

Here in this event handler for a button click, we mix validation (the two TryParse calls), business logic (v1 + v2), and presentation (reading from value1 and value2, and setting a currency formatted result). In such a simple case, it does not seem like a big deal to mix these tasks together. The issue, however, is that most applications are not this simple. But even in this case, we can have issues. Changing the textbox output to a label, or changing the operation from addition to subtraction, or adding validations all require this code to change. If you are the only programmer in charge of all aspects of the application, again, it does not seem like a big deal. However, in real applications, you will be working in a team with divided responsibilities. Coordination between two or three programmers all trying to modify the same few lines of code quickly becomes a difficult and time-consuming issue.

What MVVM will do for us is, on the surface, more complex, but by separating the various tasks into their own classes, will make management over larger projects easier. That is why this same approach is used not just in desktop Windows applications, but in most modern web frameworks, such as VueJS, React, and Angular. The syntax and naming conventions are different in each of those, but the concepts are the same.

The elements

Model

A Model is a Class or set of Classes that represents the data being manipulated by our application. For example, an application for selling cars would likely have a Car class with properties like make, model, license number, odometer reading, etc.

View

We have already been creating Views. The MainWindow class and the XAML associated with it is a View. This is the part that directly interacts with the user and encompasses the text being displayed, the formatting of numbers, and the layout of the controls.

ViewModel

A ViewModel is a Class that intermediates between the View and the Model. It contains properties and methods that reflect business logic as applied to the Model and expose those properties for use by the View.

Implementation

To successfully structure your project to use MVVM, you must, at minimum, implement the following three steps.

First, create a new file that will contain our ViewModel class. In Visual Studio, you do this by:

- Right click on the project
- Click Add...
- Click Class...
- Change the name to VM (class names should all start with capital letters)
- Click Add
- This should add a new file to the project and open it in a new tab in Visual Studio

Now, add the following code to the VM class. Most of this is 'boiler plate code'. That is, code that you really do not need to memorize, but is repeated every time, as follows:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace SampleApp
{
    internal class VM : INotifyPropertyChanged
    {
        #region PropertyChanged
        public event PropertyChangedEventHandler PropertyChanged;

        private void onChange([CallerMemberName] string property = "") =>
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(property));
        #endregion
    }
}
```

INotifyPropertyChanged

Just like MainWindow inheriting behavior from Window, our VM class inherits from INotifyPropertyChanged. Unlike Window, INotifyPropertyChanged does not provide any behaviors, instead it is a contract (an interface) that guarantees the class will provide certain attributes, specifically PropertyChanged. As you can see from the data type, PropertyChanged is an event. What we are doing here is allowing other code to register their interest in being called when a property change event occurs. This is the same mechanism that we have been using to register for Click events on Button controls, except in reverse. Instead of our code registering for someone else to let us know when an event has occurred, others are registering with us and we are responsible for letting them know.

The using statements that we need are not provided by VS by default. Visual Studio can help with those, but we need to add them ourselves.

We use the `#region/#endregion` syntax to tell Visual Studio that this is a range of code that we'd like to collapse to a single line so it is out of the way visually. You can create as many regions in your code as necessary to hide the boilerplate code unless you want to look.

We see our first instance of attributes in code with `[CallerMemberName]`. This is a modifier on the method's parameter. There are several similar helpful attributes, but this one behaves as follows: if the parameter is not provided and the default is to be used, then replace the default with the name of the calling method. We will be using it here to save ourselves a bit of typing.

We see the `?` used in another context. In this case, it checks to see if `PropertyChanged` is null and only if it is not, does it proceed to the `Invoke` method call. We need to call the `Invoke` method on the event because multiple bits of code could have registered to be notified. `Invoke` takes care of calling each event handler method that was registered. We check for null in case no code has registered.

The `onChange` method is something we need to call whenever some property changes in our VM class because we want them to know so they can react appropriately to the change.

Adding Properties

Add a property of our own to the VM class, like so:

```
internal class VM : INotifyPropertyChanged
{
    private decimal someNumber = "";
    public decimal SomeNumber
    {
        get => someNumber;
        set { someNumber = value; onChange(); }
    }
}
```

Notice that the property is written with the longer form syntax. That is because we want to add a call to `onChange()` whenever the set – a change is being made to the property – is invoked. The result is that whenever this public property is modified, `onChange()` is called. `onChange`, in turn, checks to see if any code needs to be notified that a change has taken place, and, if so, invokes their event handler code. Also note that although I could have written `onChange("SomeText")`; I did not need to because the `CallerMemberName` attribute will set that parameter for me.

However, it can sometimes be handy to use the `onChange("PropertyName")` version of this call. This is especially true if the property affected by some calculation is a get; only property. Since it has no set, there is no place within the property to call `onChange` and the call must be made elsewhere when the `CallerMemberName` attribute would return the wrong result. One detail, though, is that rather than using the string version of the property's name, you should write:

```
OnChanged(nameof(Total));
```

Using the `nameof()` expression might seem redundant, since in the case above `nameof(Total)` returns the string `"Total"`. However, the advantage of this approach is that the rename tool in Visual Studio will rename variable in the `nameof` version, but will leave the string untouched. Since the conversion to the string is done at compile time, it is no slower to run.

Linking to View

Next, we need to add code to connect the ViewModel class to the View. We do this in `MainWindow.xaml.cs` with this code:

```
VM vm = new VM();

public MainWindow()
{
    InitializeComponent();
    DataContext = vm;
}
```

Here we are doing two things. We are creating an object of the type VM using the `new` keyword. This instance of the class is then assigned to the `DataContext` property of the `MainWindow` object. Assigning the `vm` object to `DataContext` does at least a couple of things. First, it lets the `MainWindow` XAML know where to find data to display. Second, if the VM class implements `INotifyPropertyChanged`, it registers itself with the object for property changed events.

View Binding

Finally, we modify the XAML to use properties from the VM class. We do this using the *Binding* keyword, like so:

```
<TextBox Text="{Binding SomeNumber}" />
```

In this case, we bind the `SomeNumber` decimal property from our VM class to the `Text` attribute of the `TextBox` control. What will happen is that the XAML will use the get of the `SomeNumber` property to determine its current value and display it in the `TextBox` control. If a user were to type into the `TextBox`, the XAML would update the `SomeNumber` property using the set. This, in turn, triggers the `onChange()` method and the XAML then knows to update the display to reflect the newly typed text.

This example binds to a `TextBox` control's text attribute. However, it is possible to bind any VM class property to ANY attribute if the data type can be converted to match. For example, it would be possible to bind to a control's visibility, background color, or opacity.

The name specified in the binding must match the name of the property exactly. If it does not, then the XAML will just ignore the binding – no error will be generated. This is one of the advantages and disadvantages of loose coupling. On the one hand, someone designing the XAML may not know the exact names and it does not matter. They can proceed through the design independently of the VM class. The downside, of course, is that you get no error messages when you do want them.

It is possible to bind multiple controls to the same VM class property. A `Label` control and a `TextBox` control could both display the same information if the visual designer requires it, but the developer of the VM class only needs to provide that information once. The `onChange()` method in the VM class connects the two controls.

Advanced

Binding to Numeric Properties

Note that the property was declared in the VM class with the data type that the business logic wants to use, in this case a decimal. The XAML will automatically validate the user's data entry to ensure it matches the data type.

One downside of this approach is that the user can still enter a non-numeric text into the TextBox control. A relatively simple approach to dealing with this problem is to use event handling to filter out unwanted characters. To do this, we will handle the PreviewTextInput event by adding a new event handler definition to the XAML:

```
<TextBox Margin="10" Text="{Binding SomeAmount}"
        PreviewTextInput="TextBox_OnPreviewTextInput">
</TextBox>
```

And adding this code to the event handler:

```
private void TextBox_OnPreviewTextInput(object sender, TextCompositionEventArgs e)
{
    var r = new Regex(@"[\d\.]");
    e.Handled = !r.IsMatch(e.Text);
}
```

There are several new language features in this code. First, the second parameter of the event handler is different from what we received (and ignored) when handling a button click. In this case, the second parameter provides us with some information that we need – the character the user typed can be found in the Text property. What we want to do is to ensure that the character typed is a digit or a decimal point.

Regular Expressions

To do that, the code uses a built-in class for Regular Expressions. It is said that if you try to solve a problem with a regular expression, you now have two problems. And, in complex cases, that may be true. However, our case is pretty simple. Our regular expression is the text: `[\d\.]`. Note that regular expressions use the `\` character to indicate that the next character has special meaning. This is also true for C# string. Putting the `@` character in front of the string ensures that C# does not interpret the `\` character in that way.

What this text means is: the character in the string must be one of the things specified in the list contained in `[]`. The two things it could be are any digit (`\d`) or a decimal symbol (`\.`). Once we've created the regular expression object, we can use its `IsMatch()` method to check `e.Text`. If it returns true, it is a match. For more complex regular expressions, I recommend a site like 'Regular Expressions 101' (<https://regex101.com>). There you will find an easy way to interactively experiment with regular expressions. Of course, since this regular expression only tests the character being typed, it is still possible for the user to attempt to type two decimal characters. This would still pass our test, but would not be a valid number. But perhaps we can hope users are not trying to break things intentionally.

Finally, we are setting `e.Handled` to be the inverse of whether there was a match or not. What the `Handled` property does is tell the event invoking code whether to continue invoking event listeners or to stop. If we say that we have handled the event (`e.Handled = true`), then no other event handler will be

notified that the event has occurred. If you will recall from the PropertyChanged discussion, multiple bits of code could be listening for an event to occur. The order in which they are invoked is in reverse order of registration. So, our code is processed earlier than system level listeners. That way, our code runs, determines that no further processing is required and the built-in listeners do not handle the character and it does not show up in the text box.

Validation of Numeric Input

However, this still leaves two problems. The double decimal issue I mentioned above is one. The other is due to an artifact of the binding between XAML and our ViewModel class. The XAML only updates the bound property on valid data. If the user enters valid data, deletes it, and then enters invalid data, the display and the property will be out of sync. This could cause invalid results from calculations depending on the application. To deal with these issues, we need to write some more complex code and handle a different event.

Instead of looking at one character at a time to determine only if the character is valid, we really want to look at the entirety of what has been typed so far and judge whether that is valid or not. We can do this with the TextChanged event and code something like:

```
private decimal? previousValue;
private void TextBox_OnTextChanged(object sender, TextChangedEventArgs e)
{
    if (sender is TextBox tb && !string.IsNullOrEmpty(tb.Text))
    {
        var r = new Regex(@"^[0-9]*\.[0-9]*$");
        e.Handled = !r.IsMatch(tb.Text);

        if (!e.Handled)
            previousValue = decimal.Parse(tb.Text);
        else
        {
            vm.SomeAmount = previousValue;
            tb.CaretIndex = tb.Text.Length;
        }
    }
    else
    {
        vm.SomeAmount =
            previousValue = null;
    }
}
```

Again, we have some new syntax to discuss. First new thing is the test:

```
sender is TextBox tb
```

This code takes the first parameter passed to our event handler – that we have ignored so far and tests to see if it is a TextBox control. It will be, since the only event this code is attached to in our case is the TextChanged event for this TextBox control. In addition to testing, it assigns the contents of the sender object to tb – a variable of type TextBox. Or, if sender is not a TextBox control, then it will assign null to tb and the result of the test is false. We pair that up with some validation code that sets the SomeAmount property to null if either are not true.

Next, we use a regular expression again. This time, however, we are looking at all the text in the textbox, so we need a more complex regular expression:

```
@'^[0-9]*\.[0-9]*$"
```

This regex can be broken down as:

- ^ - from the beginning of the text
- [0-9]* - look for zero or more digits
- \. - look for zero or one decimal characters
- [0-9]* - look for zero or more digits
- \$ - to the end of the text

The result being that anything with non-digit characters will fail, anything with more than one decimal character will fail, but the strings "12", "12.", "12.12", ".12" will all pass.

If it passes, we want to remember the value – so we parse the text and save the resulting decimal number. If it does not pass, then we want to restore the previously good decimal value and set the caret (the vertical bar indicating the text insert point) to the end of the string, since updating `vm.SomeNumber` will set the caret to the start of the text.

Note that we do not overwrite `tb.Text`, but that we update the decimal property of the VM class. If you overwrite `tb.Text`, then you are removing the binding and replacing it with a fixed value and updates will stop.

Formatting Numeric Properties

To format the text in a `TextBox`, you can use the `StringFormat` attribute, like so:

```
<TextBox Text="{Binding SomeNumber, StringFormat=C}" />
```

In this case, the text will be formatted as currency for display, while the VM class property remains a decimal data type for calculations without needing to be parsed.

Formatting text in a `Label`, however, is somewhat different, but it can be done this way:

```
<Label Content="{Binding SomeNumber}" ContentStringFormat="C" />
```

Property Updating Modes

By default, the View will not update the ViewModel with new data for the property until the control has lost focus. Generally, this occurs when some other control, like a button, is clicked. You can change this behavior with the `UpdateSourceTrigger` attribute set to `PropertyChanged`. This will update the ViewModel property on every user modification, and is requested like so

```
<TextBox Text="{Binding SomeText, UpdateSourceTrigger=PropertyChanged}" />
```

With this mechanism in place, any changes the user makes to the `TextBox` will be reflected in the VM property AND any changes to the VM property done via calculation will be reflected in the UI of the application. All without you needing to write additional code to ensure they are kept in sync. However, this setting does cause us one problem with our input validation code. We can no longer type a decimal character.

The reason this is happening is that the default behavior is for the text in the TextBox and the text version of the decimal value to match. Since converting the string to “12.” results in decimal value of 12, the conversion back to a string ends up as “12” – without the decimal. Every time the user types the decimal, it triggers a PropertyChanged event which updates the display. Fortunately, there is a simple fix for this problem. Simply add this code to the App class in App.xaml.cs:

```
public App()
{
    FrameworkCompatibilityPreferences.
        KeepTextBoxDisplaySynchronizedWithTextProperty = false;
}
```

This changes the behavior so the TextBox control does not continually update the display to match the bound property. As a result, the decimal character stays.

Adapting to the Current Culture

One last point about all of the above is that we’ve assumed that the decimal separator character is ‘.’. Depending on your locale, this may not be true. If you want to support other locales, then use the CultureInfo built in object:

```
var sep = CultureInfo.CurrentCulture.NumberFormat.NumberDecimalSeparator;
```

to extract the separator, and many other date and number characters. CurrentCulture will reflect whatever the computer is set to by the user.

Binding Modes

Note that if you want to limit the binding to updating in only one direction, you can use the Mode attribute of Binding to set something other than the default, like this:

```
Content="{Binding SomeNumber, Mode=TwoWay}"
```

The options are:

- OneTime – the initial state of the property is reflected in the displayed value and not subsequently updated
- OneWay – changes in the property are reflected in the displayed value
- OneWayToSource – user changes in the XAML are reflected in the ViewModel property
- TwoWay – both OneWay and OneWayToSource
- Default – the default is TwoWay for editable controls and OneWay for display only controls

ValueConverter Interface

At times, we will want to represent attributes like visibility in our VM. To keep the VM isolated from the specific implementation of a particular View, we should NOT represent Visibility based on its implementation for XAML controls (Visibility.Hidden, Visibility.Visible). Instead, we should represent it in the VM as a boolean – usually true for visible and false for hidden. The problem with this is that we cannot directly bind a Boolean to the Visibility attribute in the XAML – they are not the same type. Instead, we need something on the View side to translate between the VM’s boolean and the View’s Visibility enum. Such a thing exists in the form of the *IValueConverter* interface. Specifically, for our example, there is a built-in class called BooleanToVisibilityConverter that will do this work for us.

To add a reference to the built-in converter, you need to add the following to the Window.Resources tag of your XAML:

```
<BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
```

And then, to use the converter in a particular control, add the Converter attribute to the binding, like so:

```
<Label Visibility="{Binding HasNumber,
    Converter={StaticResource BooleanToVisibilityConverter}}" />
```

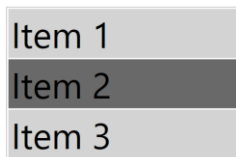
The only other interesting built-in converter is the AlternationConverter. Use this converter to create alternating background colors in a ListBox control. By adding this to Window.Resources:

```
<AlternationConverter x:Key="BackgroundConverter">
    <SolidColorBrush>LightGray</SolidColorBrush>
    <SolidColorBrush>DimGray</SolidColorBrush>
</AlternationConverter>
<Style x:Key="altBgd" TargetType="{x:Type ListBoxItem}">
    <Setter Property="Background"
        Value="{Binding RelativeSource={RelativeSource Self},
            Path=(ItemsControl.AlternationIndex),
            Converter={StaticResource BackgroundConverter}}"/>
</Style>
```

which applies one of two colors to the background property of a ListBoxItem. Then modify the ListBox attributes like so:

```
<ListBox AlternationCount="2" ItemContainerStyle="{StaticResource altBgd}">
```

to produce a visual result like:



Creating your own ValueConverter

In addition to using the built-in converter classes, you can build your own converter for application specific things (like converting a VM error code to text for display) by creating your own class and implementing the IValueConverter interfaces. For example, you may want to convert something like our EggSize enum into displayable text. Recall that the enum looks like:

```
public enum EggSize
{
    PeeWee,
    Small,
    Medium,
    Large,
    ExtraLarge
}
```

We want to use the enum in our VM code because it will be easier to write and debug the code with a fixed set of values in an enum. However, we also want to show the user some text to describe the size of

the egg and not an integer they would be required to interpret. To accommodate this mismatch, we create a class like this:

```
class EggSizeToTextConverter: IValueConverter
{
    public object Convert(object val, Type type, object p, CultureInfo c)
    {
        var result = string.Empty;

        if (val is EggSize eggSize)
        {
            result = val switch
            {
                EggSize.ExtraLarge => "The egg is extra large size.",
                EggSize.Large => "The egg is large size.",
                EggSize.Medium => "The egg is medium size.",
                EggSize.PeeWee => "The egg is classified as peewee sized.",
                EggSize.Small => "The egg is small.",
                _ => "The egg is of unknown size."
            };
        }
        return result;
    }

    public object ConvertBack(object val, Type type, object p, CultureInfo c) => null;
}
```

We make this class use the `IValueConverter` interface. In this case, this interface requires us to implement two methods: `Convert()` and `ConvertBack()`. Actually, we'll never have the user type in some text to describe the egg size and need to convert that back into the enum, so we can safely leave `ConvertBack` unimplemented. In the `Convert()` method, we want to map the enum to some unique text for each possible value. The enum value arrives to this method in the first (`val`) parameter. In this case, we don't care what is provided in the other parameters. If `val` is indeed an object of `EggSize` type, then we can use the switch expression to map the enum to a string value and return that. Note that the last parameter is a `CultureInfo` object. We can use this to provide translations of the text as needed.

Now that we have created a converter, we need to tell the XAML about it by adding this to `Window.Resources`:

```
<local:EggSizeToTextConverter x:Key="EggSizeToTextConverter"/>
```

Because this is our own custom converter, we need to tell the XAML where to look for it. To do that, we prefix it with "local". Visual Studio defined this for us as the "xmlns:local" attribute of the `Window` tag. Now that we have it accessible in the XAML, we can use it like this:

```
<Label Content="{Binding Egg.EggSize,
                        Converter={StaticResource EggSizeToText}}"></Label>
```

Setting the `Converter` attribute of the `Binding` will cause the `EggSize` property of the bound `Egg` object to be passed to the `EggSizeToText` converter and then the result of that conversion will be displayed in the `Label` control.

Advanced Validation

If the validation behavior provided by default is insufficient, you can add your own error display on invalid data. Doing this requires a new class that inherits from the `ValidationRule` class and some additional component to change the display if an error is detected. For example, a validation rule class might look like:

```
public class AgeRangeRule : ValidationRule
{
    public int Min { get; set; }
    public int Max { get; set; }

    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        var vr = ValidationResult.ValidResult;

        if (value is string v && int.TryParse(v, out var age))
        {
            if (age < Min || age > Max)
                vr = new ValidationResult(false,
                    $"Please enter an age in the range: {Min}-{Max}.");
        }
        else
            vr = new ValidationResult(false, $"Invalid input");

        return vr;
    }
}
```

This time, we are inheriting behaviors of the `ValidationRule` class, so rather than implementing some required interface, we use the 'override' keyword to replace the default implementation with our own custom implementation of the `Validate()` method. Just like the converter interface, we will receive the value the user typed and then do our checking. We return a `ValidationResult` object.

To use this class, we must add it to our `Window.Resources` tag:

```
<local:AgeRangeRule x:Key="AgeRangeRule"/>
```

And then refer to it in the XAML:

```
<TextBox Name="AgeText">
    <TextBox.Text>
        <Binding Path="Age" UpdateSourceTrigger="PropertyChanged" >
            <Binding.ValidationRules>
                <local:AgeRangeRule Min="21" Max="130"/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
<Label
    Visibility="{Binding ElementName=AgeText,
        Path=(Validation.HasError),
        Converter={StaticResource BooleanToVisibilityConverter}}"
    Content="{Binding ElementName=AgeText,
        Path=(Validation.Errors)/ErrorContent}"
></Label>
```

Notice that the binding syntax here is different from what we'd previously written. However, all we are doing is replacing the attribute version of the binding with an xml element version of the binding. We do this to give us the ability to specify the `ValidationRules` tag as a child of the `Binding` tag. Also note the relationship between the public properties `Min` and `Max` and how they are set in the XAML.

The `Label` control is added to show the error message if something is wrong with the input. Here we are binding to the properties of another control on the page. To do that, we use the `ElementName` syntax to specify which element on this page we want to connect with and the `Path` keyword to specify the particular property. In this case, `Validation.HasError` is a boolean that indicates whether there is an error. `Validation.Errors/ErrorContent` contains the actual text of the error message. We can bind those to the `Visibility` and `Content` attributes of the `Label` control to have it show up only if the input is in error.

Use this approach if, in your situation, validation is considered part of the UI. For instance, you may have a VM class that does some calculation regardless of age and is used in many situations. But this particular UI is targeted at only a certain age group. In that situation, you will want to separate the age range validation from the business calculations. If that's not the case and the validation is done within the VM class, then using a value converter class to convert error codes to text strings is sufficient.

Command interface

There is one more step we could take, and that is to use the *Command* interface to eliminate any code associated with button click event handlers. The one advantage this approach has is the ability to enable/disable buttons based on VM properties. However, in software I design, I always allow the user to press buttons and then display a message to indicate why the action cannot be performed. I find this to be more user friendly than disabling the button for reasons that may not be obvious to the user. As well, the `ICommand` approach requires significantly more code, so I do not do it. You can review this document: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/commanding-overview> to decide for yourself.

Exercise 6

- Rewrite Exercise 4 using the MVVM approach. Once completed, compare your version with Exercise 6.

Loops

To this point, the applications we have created only required that a decision be made at some point to branch to one block of code or another. And, for some applications that is sufficient. However, many applications require that we repeatedly execute some block of code until some boundary condition is met. For that reason, all programming languages have a way of expressing the ability to repeat a set of instructions for some number of times.

Although it is possible to require a loop that never ends, most often loops are intended to terminate at some point. If you accidentally create a loop in your application that does not terminate, then you have what is known as an “infinite loop” error and the only way to stop it is to use Task Manager in windows to kill the application.

The C# language has four different approaches to loops. Three of those will be introduced here and the fourth will be discussed in the next section. The three discussed here have some common elements: an initial value stored in a variable, a terminating condition associated with that variable, and a way of modifying that variable so that it eventually ends up at the terminating condition.

While

The basic structure of a while loop is like that of an if statement:

```
<initialize a variable>
while (<some statement checking the variable that evaluates to a boolean>)
{
    <a series of statements to be executed only if the boolean is true>
    <a statement modifying the variable>
}
```

Anything you could write in an if test, you could also write in the while test. The difference is that if the statement evaluates to true, the code will repeat. Occasionally you will see naïve code written using while where if would also have worked. Although technically correct, this is considered bad form and should not be done. If you know that the loop will only ever be executed once, use an if.

In C#, you would write:

```
var x = 0;
var s = string.Empty;

while(x < 7)
{
    s += x.ToString();
    x++;
}
```

This code initializes a variable ($x = 0$) that we will use for determining when the loop is to complete. The code within the following `{}` will be executed repeatedly as long as x is less than 7. To ensure that we do not spend forever executing this loop, we increment x by 1 ($x++$).

Note that there are several ways of incrementing x by 1. You could write:

- $x = x + 1;$
- $x += 1;$

- `x++;`

If you use any way other than `x++`; you will be found out as a newbie programmer. No experienced developer would ever use any other approach – it is all about efficiency and the least typing possible to get to the action we want.

We also could have written the string concatenation as `s = s + x.ToString()`; but again no experienced developer would ever write it that way for the same reasons.

When the condition turns false and the loop terminates, the variable 's' contains "0123456" or seven characters worth of data.

In C#, it is expected that you will start the loop counter variable at zero and then have the condition be `<` with the comparison value being the number of times the loop code will be executed. Change from this pattern only if there is some very good reason to switch.

One of the most common errors in coding is the "off by one" error where the loop is executed one more or less time than you expected. If you stick with `x < 7`, you will minimize the chance of introducing this error.

While loops evaluate the condition at the start of each loop. So, when x is incremented to 7 at the bottom of the loop, the next go around first tests to see if x is less than 7 and this is no longer true, so the 8th time stops the loop.

To create an infinite loop, simply remove (or forget to add) the `x++` statement. Without it, the test will never be false since zero is always less than seven. Eventually, the computer will run out of memory to store this increasingly long string. Since strings are immutable and the new string is a copy of the old string plus the new characters, that point would come when the string gets to the length of approximately half the size of available memory.

For

The for loop has all the same components as a while loop, but organized differently. For example, the above while could be rewritten as:

```
var s = string.Empty;

for(var x = 0; x < 7; x++)
    s += x.ToString();
```

The interesting part of a for loop is that inside the () the for contains three separate statements. The first statement sets the initial conditions. The second statement sets the test condition. The third statement defines the mechanism for updating the test, so it eventually stops.

These three statements are the same as the three individual statements in the while case. The advantage of writing it this way is that the statements are all in one place rather than scattered through the code. It makes it less likely that you will forget the adjustment line and cause an infinite loop. For this reason, the for loop is preferred in most cases.

You are allowed to provide empty statements to any of the three components of the for loop, which means that `for(;;)` is valid syntax. And this is how you would create an infinite loop.

Do/While

There are times when you know you want to execute a block of code at least once, but then perhaps more times. In those cases, a do/while loop is appropriate.

The syntax for do/while looks like:

```
var x = 0;
do
{
    s += x.ToString();
    x++;
}
while (x < 7);
```

In this case, the do/while version generates the same result as the for and the while version above. But, if the comparison were $x < 7$, but $x < 0$, this version would add one character to *s* where the other two would not.

When to use each type of loop

You may need to use the do/while when writing code that attempts to send a message over a network connection. You know you want to send the message and it may go through on the first attempt. However, it is possible that the network is not connected, and repeated attempts must be made. A do/while would allow you to check to see if the message had been sent at the end of the loop and try again if something went wrong.

I recommend using the while loop only in the situation where the number of iterations required through the loop is unknown ahead of time. In our straightforward case, we are always incrementing *x*. But if the logic required that *x* only be incremented sometimes, then a while loop is appropriate. As we'll see later, reading data from external sources, such as files and databases, is also a good time to use the while. Again, we have no idea ahead of time how much data we will need to read from these sources.

Populating a ListBox

Now that loops are in our toolkit, populating a ListBox is a much simpler task, at least for repeating items.

For example, we can write:

```
for (int x = 0; x < 7; x++)
    SelectList.Items.Add($"Item {x + 1}");
```

Notice that the user of the application will see "Item 1", "Item 2", etc. Normal people do not like counting from zero, so I have labeled the items starting at 1. However, as programmers, we DO want to count from zero, so internally I have written my code in the normal way and adjusted only the display.

This is a general principle in programming: keep your internal representation for as long as possible and only convert to something you want to user to see at the last possible moment. Also, in reverse, convert new data from the user to its internal representation as soon as possible and work with it like that the rest of the way.

Creating a control in C#

We have been using XAML to create and lay out the controls in our UI and this is the typical approach. However, there are times when this approach can be quite tedious and potentially error prone. For instance, the UI for a Sudoku game requires a 9x9 grid of cells. However, if you want to provide users with the ability to note the possible numbers that can go in each cell, that requires 9x9x10 or 810 controls. Not something you would want to type manually, even with cut and paste.

Fortunately, it is possible to use C# to create controls and place them. If we had a Grid control already in our window, we might want to add buttons and place them somewhere on the grid. To create a Button control, you would write something like:

```
var gridButton = new Button()
{
    Content = "Button Text",
    Margin = new Thickness(1),
    Tag = index,
};
```

This creates a button with some content and a margin. It is, however, missing a click event handler, which we can add by:

```
gridButton.Click += gridButton_Click;
```

Again, we are adding our event handler code to the list of handlers that will be invoked when this button is clicked. If we created more than one button in a loop, we could still assign the same button event handler to all the buttons created because of the “Tag” property above. As long as you assign a unique value to each button in the Tag property, you will be able to determine which button has been pressed. The Tag property takes an object, so you can assign any simple or even an instance of a class you define.

Now that we have a button, the next step is to set its position. In XAML we would say Grid.Row= and Grid.Column=. In C#, the equivalent is:

```
Grid.SetRow(gridButton, row);
Grid.SetColumn(gridButton, col);
```

Finally, we must associate this button with a specific grid. We do that by adding the button as a child of the named Grid control:

```
NamedGrid.Children.Add(gridButton);
```

If we did not want to have a fixed number of rows and columns in our grid, we can also set those programmatically by:

```
for (var row = 0; row < NUM_ROWS; row++)
    NamedGrid.RowDefinitions.Add(new RowDefinition());
```

Notice that here I use the for loop approach, since I know ahead of time how many rows I will want.

Exercise 7

Create a checkerboard inside a Grid control by using loops in C# to create button controls and place them in the Grid. Have each of the buttons respond to a click by adding a ListBoxItem to a ListBox that

indicates which button was pushed. Once completed, compare your solutions with the code found in Exercise7.zip.

Writing to Files

There are several approaches to writing files in C#. However, the simplest (and therefore recommended) approach for most is to use the System.IO.File library. It is not in the list of using statements at the top of the file Visual Studio creates for you by default, so you will need to add this to the using section of your file:

```
using System.IO;
```

Once that is present, you will be able to write something that looks like this:

```
File.WriteAllText("filename.txt", "contents");
```

To write to a file. The first parameter specifies the path, name and extension of the file. The second parameter is one long string containing everything you would like to write to that file. Writing one long string to a file can be a problem, though, because normally text is broken up into sections by line breaks. Fortunately, we can add line breaks wherever we want with a few different approaches. In this case, the line break character is embedded in the string:

```
File.WriteAllText("filename.txt", "first line\nsecond line");
```

The `\n` is called a newline character and will insert a line break character into the text. If you open this file with a text editor, you will see “first line” on line one and “second line” on line two on the display.

A couple of things about the `\` character. This is known as an escape character. It doesn’t work by itself, but ‘escapes’ the next character to create an escape sequence and give it a special meaning. We have encountered this before with regular expressions and the behavior is the same. However, the valid set of characters that may be escaped is different for files. In the case above, the `n` means newline. There are several other escape sequences that are commonly used in text files:

- `\t` inserts a tab character in the string
- `\r` inserts a return character in the string
- `\\` to display a `\` character in the string
- `\"` to actually display a double quote character in the string

There is disagreement about exactly what characters indicate a newline. Unix and the web use `\n`. Windows uses `\r\n`. The Mac originally used `\r`, but now that its underpinnings are a Unix version, it has switched to `\n`. Many Windows applications now handle both, although there are still instances where an application will not display a line break with just `\n` in the text.

Now that .NET is available for all three platforms, the safest approach is to avoid your own logic around `\n` and use a property built into .NET, like so:

```
File.WriteAllText("filename.txt", $"first{Environment.NewLine}second");
```

`Environment.NewLine` will take care of determining the platform on which your code is running and insert the appropriate sequence for a line break.

If you specify the file with no path information (as in the previous examples), the file will be written to the ‘current’ directory. To determine ‘current’, just find your .exe file and the file you asked to be created will also be there. For testing purposes, this is not a problem. However, once the application is

published, this approach will not work. Applications are installed to the C:\Program Files directory and that directory can only be written to by Windows itself. This is to protect your application from malicious applications that, in the past, would replace your executable files with either own pretending to be yours. Instead, then, Windows provides a separate directory for applications to write private data that is not to be shared directly with the user. To access this directory, write:

```
var path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);  
var fullName = Path.Combine(path, "filename.txt");
```

There are various special folders in Windows, but the application data folder is where applications can store private data. The path itself is C:\Users\<user>\AppData\Roaming. This is a hidden folder, so normal users will not know it is there. If you changed the File Explorer settings as I recommended, however, you will be able to see this folder and navigate to it. If you do, you will see every app you have installed will be represented with a folder.

The code above puts the file directly in the Roaming folder. To be a good citizen, however, you should create a subdirectory for your company or application and put your data there. To do that, we need to add an extra step:

```
const string DIRNAME = "TestApp";  
var path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);  
path = Path.Combine(path, DIRNAME);  
if (!Directory.Exists(path))  
    Directory.CreateDirectory(path);  
string fullName = Path.Combine(path, "filename.txt");
```

The code above adds a subdirectory to the path where we can store information specific to our application. Actually, if you notice the structure of the Roaming folder, it is more typically company name folders and then, within those folders, additional folders for specific applications. Of course, wishing that the directory exists will not make it so. The code above checks for the existence of the full path and if it does not already exist, creates it.

Do not use string concatenation to build a path. Using Path.Combine() means you do not need to concern yourself with the correct placement of the path separator character (which can be different in various cultures). It also takes care of removing double path separators and adding missing path separators. Since the code exists to deal with all the possible cases, take advantage and do not write your own.

If, instead of private files, you do want to allow the user to pick the location of a file to write, there is a standard mechanism provided by Windows:

```
var ofd = new OpenFileDialog();  
if (ofd.ShowDialog() == true)  
    File.WriteAllText(ofd.FileName, "contents");
```

You would have used the OpenFileDialog many times in Windows. Almost all programs use it. If the user selects a directory and file name and clicks OK, the ShowDialog method will return true and the FileName property will contain the user's choice for file location. Use that directly in the WriteAllText() call.

One thing about using `WriteAllText()`. Each call replaces the previous contents of the file with the new contents. This may be what you want, but usually what is actually desired is to append the text to the current contents of the file. To do this, use `AppendAllText()` instead.

Finally, to be most efficient when using `AppendAllText()` or `WriteAllText()`, you would build your string using the `StringBuilder` class and then, when the string is complete, convert it to a string object and use the appropriate method to save it to a file.

Exercise 8

Take your checkers application from Exercise 7 and add a new method to update a file with the information about the cell the user clicked on. Write this file to an app specific directory in `AppData`. When completed, compare your version with what is found in Exercise 8.

Collections

To this point, we have been using variables that hold one value. As you have probably noticed, it would be convenient at times to have a group of values that we can refer to by a single name. Applications that deal with the set of all employees, or the line items on an invoice, or the lines of text in a file, or even the words on a line are all more simply expressed if we do not need to create a separate variable for each thing.

In C#, most of the classes that organize a set of like objects into a single entity inherit from the `Collection` class. You can tell a collection because of the `[]` characters that are traditionally used to indicate collections – usually with an index value in between. An index into a collection specifies which of the elements in that collection we want to refer. Most of the time, the index is an integer, but it can be other data types as well.

Array

The simplest way to group elements together is the array. The array is not actually a collection. Historically, C and C-like languages have all had array data types and collections were implemented in a separate library, or the programmer implemented needed collection methods on their own. However, C# makes both arrays and collections equally available, so they can be treated similarly in your code.

When you declare an array, like so:

```
var stringArray = new string[12];
```

You are specifying that you would like to allocate space in memory to remember the locations of 12 strings. You are NOT specifying the strings themselves.

An analogy I like to use is that of an egg carton. The statement above is the equivalent of creating an egg carton. But, just as creating an egg carton does not automatically mean that there are eggs in it, creating `stringArray` does not automatically mean there are strings in it. What is there is the null value, so your code needs to take that into account. Note that this is different behavior from JavaScript. It is easy to confuse the two since the syntax is very similar, so be careful.

To populate the array, you write something like:

```
stringArray[7] = "element 7";
```

Or, using our knowledge of loops, write:

```
const int ARRAY_SIZE = 12;
string[] stringArray = new string[ARRAY_SIZE];
for (var i = 0; i < stringArray.Length; i++)
    stringArray[i] = $"element {i+1}";
```

Notice that I used a constant instead of a magic number to specify the size. Also, that I used the `.Length` property of `stringArray` rather than the constant for the loop termination test. So even though I am using a variable to determine the number of times through the loop, it is still known to me ahead of time so I use the `for` rather than a `while` loop construct.

The reason for using the `Length` property is because in more complex applications, it is possible to assign a new array to the same variable. That new array may have a new, shorter, length. In that case, using

the constant would cause .NET to throw an exception. Accessing using an index that is outside the bounds of the array will always throw an exception in C#. That is true if the number is negative as well as if it is larger than the defined size of the array. It is a good idea to get into the habit of using the array's own stated length rather than a constant, so this does not happen.

If you know what you would like to put into the array from the start, then you can use this:

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

Notice we no longer need a constant to specify the size of the array. The array is sized to exactly the number of elements inside the {}.

Or we could write even more simply:

```
var intArray2 = new [] { 1, 2, 3, 4, 5, 6 };
```

Here we do not even specify the type of the array because C# will infer that since the initializing data is of type int, then the array must be an int array with 6 elements in it. Although it is possible to create an array of object and then put any data type into any element of the array, this is generally not considered appropriate in C#.

To read from an array, put the array and its index on the right side of the assignment sign, like so:

```
int x = intArray[4]; //retrieves the value 5 (counting from 0)
```

Note that just like the assignment, if we were to specify an index larger than the array, it would throw an exception.

Arrays may have more than one dimension. A two-dimensional array can be declared this way:

```
int[,] int2d = new int[3, 3];
```

And initialized this way:

```
int[,] int2d = new [,] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Or this way:

```
int2d[0, 0] = 1;
```

The first index indicates which group, the second index indicates which member of that group. And both indices count from zero. So,

```
int2d[1, 2] == 6
```

is true.

It is also possible to create an array of arrays, or Jagged array, like this:

```
int[][] intJagged = new int[3][];  
intJagged[0] = new[] { 1, 2, 3 };  
intJagged[1] = new[] { 4, 5, 6, 7 };  
intJagged[2] = new[] { 8, 9, 10, 11, 12 };
```

In this case, we have an array of three arrays and the first of them has been initialized with 3 values. The other two have 4 and 5 values. Because each array is independent, they can be different lengths.

To access an array element, you use:

```
var is6 = intJagged[1][2] == 6; //this is true
```

It is also possible to initialize an array of arrays like this:

```
var intJagged2 = new int[][] { new [] { 1, 2, 3 },  
                               new [] { 4, 5, 6, 7 },  
                               new [] { 8, 9, 10, 11, 12 } };
```

Because each array within the array is independent, we need to create a new array for each element, thus the use of the new keyword.

List

Arrays are great when you know how many entries to create ahead of time. Many times, however, you do not. When you discover that you require more or fewer elements, you could create a new array with the new size and copy all the elements from the old array to the new array. However, that is a lot of work to do each time you want to add an element and it will slow down your application.

Instead, C# provides a member of the Collections family of classes – the List class – to take care of resizing. A List is declared like:

```
var intList = new List<int>();
```

Note the new syntax using <>. This indicates that the List class is a Generic collection, meaning that it allows for the creation of a list of any data type. In this case, we are specifying that we want a list of integers. Unlike arrays, no provision of size is necessary. For efficiency, creating a list does allocate some space for more data as you add elements, but unlike the array, it is automatically increased if the default size is exceeded. If you know that you will be adding a large number of items to the list, you can provide a parameter to specify a starting Capacity, like so:

```
var intList = new List<int>(10000);
```

Note that even though you have specified a large capacity, C# will not allow you to access elements that have not yet been assigned.

Initially, lists are empty. Adding to a list is done like:

```
intList.Add(1);
```

Add as many items as you want at any time. The newly added item is added to the end of the list and can be accessed in the same way as an array:

```
var y = intList[0];
```

The simplest way to replace a list element uses the same syntax as an array. Also like an array, indexing past the end of the list will cause an exception to be thrown. But, in addition, the List other methods, some of the more commonly used are (the T syntax is just a placeholder for the actual datatype):

- `Clear()` – removes all the elements from the list
- `IndexOf(T item, int startIndex)` – returns the index of the first occurrence of the object in the list
- `Insert (int index, T item)` – adds an object at the index provided
- `Remove (T item)` – removes the matching object from the list
- `RemoveAt(int index)` – removes the object at that index from the list

A word about how `IndexOf` and `Remove` find a match. If the list contains a simple datatype, then the matching proceeds as you would expect – that is, looking for the string “xyz” will find the first match where the string contains those, and only those, characters. However, a list of complex objects does not behave this way. For example, say I have two `Egg` objects, both of which contain the same property values. If I use `IndexOf` and `Remove` passing in one will not find the other as a match. It needs to be the same object as is in the list for a match to occur. But, since complex objects are passed by reference, you will most likely have the object you need.

Dictionary

Lists and arrays are good when you want to access items in sequential order. Sometimes, though, it is important to find items based on some other criteria than the order they were added.

An example of this might be a collection of product objects for a Point-of-Sale system. Ordering the products by UPC barcode would be very useful since the most likely way of entering data is via barcode scanner. It would be possible to use `IndexOf` to search a list of products, but if the store sells 20,000 distinct items, then searching that list will take a long time, as on average, it would require testing 10,000 products before the correct one was found – since sometimes the product would be in the first half of the list, other times it would be in the second half, so on average half the list needs to be tested. However, the `Dictionary` class uses a hash table internally that requires one test to find the right object. It is not often you find an easy 10,000 time speed up for your code, so this is something to take advantage of, if needed. The one downside is that the dictionary must be populated in the first place, so the data needs to be something you will be using multiple times to make it worthwhile.

A `Dictionary` is declared like:

```
var dict = new Dictionary<string, int>();
```

Again, it is a `Generic` collection and uses the `<>` syntax to indicate the data type. Unlike the `List`, there are two types to be specified. The first type identifies the datatype of the index, or in dictionary terms, the key. The second data type indicates the type of the value. For example, initializing a dictionary like this:

```
var dict = new Dictionary<string, int>();
```

Creates a dictionary that will use a string as a key and the value returned will be an integer. You can add an item to this dictionary like this:

```
dict.Add("text", 1);
```

Or, if you have a set of known items, like this:

```
var dict = new Dictionary<string, int>() { { "a", 1 }, { "b", 2 } };
```

And access like this, using the same syntax as for an array or list, except that the index value is now the data type you specified and is not fixed to be an integer:

```
var z = dict["text"]; //returns 1 into z
```

The issue with using the syntax above is that attempting to access by an index that does not exist will throw an exception. To handle that possibility, your best choice is to access values from the dictionary like this:

```
bool isFound = dict.TryGetValue("text", out var q);
```

TryGetValue returns true if the element exists and puts the value matched with the key in the variable q. This should look familiar, as it is the pattern used by the TryParse methods.

A different exception will be thrown if you try to add an entry with a key that already exists because keys must be unique. To check for an already existing key before attempting to use Add to insert a new key, use:

```
bool exists = dict.ContainsKey("text");
```

One strategy to use if multiple values must be associated with a single key is to use a List as the value of the entry. For example, you might write this:

```
var ld = new Dictionary<string, List<int>>>();
```

Taking this approach will allow for multiple values against one key by putting all the matches in the list object associated with that key. For example, in our product dictionary by UPC code, we might have various sizes of jeans from one manufacturer associated with a single UPC code. The manufacturer gives them one UPC regardless of the size, but for our inventory counts, we need to know which size was sold.

HashSet

A third collection class is the HashSet. This is not used as frequently as the other two collection types already described, but it does provide some unique functionality that saves a lot of custom coding if it is needed. A common issue is to compare two collections of items to decide what has been added or removed and what is unchanged. Using HashSet, we would write something like:

```
var ol = new List<int> {1, 3, 4, 5, 7};
var nl = new List<int> {1, 2, 6, 7};

var removed = new HashSet<int>(ol);
var added = new HashSet<int>(nl);
//find the numbers that have not changed
var unchanged = new HashSet<int>(removed);
unchanged.IntersectWith(added);
//remove unchanged to leave only those that are no longer present in new
removed.ExceptWith(unchanged);
//remove unchanged to leave only those that have been added
added.ExceptWith(unchanged);
//unchanged: 1, 7
//removed: 3, 4, 5
//added: 2, 6
```

Compare this to code that generates the same result without the HashSet class:

```

//this algorithm assumes ol and nl are sorted
var oi = 0;
var ni = 0;
//create lists to store each category of number
var unchanged = new List<int>();
var removed = new List<int>();
var added = new List<int>();
//loop until we get to the end of both lists
while (oi < ol.Count || ni < nl.Count)
{
    //found a match?
    if (ol[oi] == nl[ni])
    {
        //add to the unchanged list and move past it in both lists
        unchanged.Add(ol[oi]);
        if (oi < ol.Count - 1)oi++;
        if (ni < nl.Count - 1)ni++;
    }
    //if the old list number is larger, then the new list number is one
    //we don't have on our list, so it has been added
    else if (ol[oi] > nl[ni])
    {
        added.Add(nl[ni]);
        if (ni < nl.Count - 1)ni++;
    }
    //if the old list number is smaller, then the old list number is one
    //we no longer have on our list, so it has been removed
    else if (ol[oi] < nl[ni])
    {
        removed.Add(ol[oi]);
        if (oi < ol.Count - 1)oi++;
    }
}
//unchanged: 1, 7
//removed: 3, 4, 5
//added: 2, 6

```

Several things should be noted in comparing these two bits of code. First, the HashSet version is much less code, so potentially easier to understand. Second, the non-HashSet algorithm requires that the two lists be in sorted order. The HashSet version did not place any restrictions on the order of the elements in the original arrays.

Finally, notice the use of the while loop. Even though the number of items in each list is known ahead of time, the number of times the loop is executed is unknown as it depends on the number of items that are the same and the number that are different. If, for example, each list has three elements all the same, then the loop is executed three times. If, on the other hand, each list has three elements that are all different, then the loop is executed six times.

BindingList

The BindingList class is a Collection class similar, but more limited than the List class. The one capability unique to it is that it will manage PropertyChanged events on its own. This means that in your VM class, you can add a BindingList as a property like so:

```
public BindingList<Egg> Eggs { get; set; } = new BindingList<Egg>();
```


without needing to create a private version and call the `onChange()` method in the set. However, you must be careful when using `BindingList`. If you were to write:

```
Eggs = new BindingList<Egg>();
```

Somewhere later in your code, you will find that the binding no longer works. That is because the XAML binds with the specific instance of the `BindingList` that was present when the `DataContext` property was assigned. The line of code above destroys that initial `BindingList` object and replaces it with one that XAML does not know about, so it cannot listen for its notification events. Instead, if you want to replace the contents of the `BindingList`, you must use this pattern:

```
Eggs.Clear();  
Eggs.Add(new Egg());
```

Note that `BindingList` only notifies the XAML when the list itself changes – that is items are added or removed. Changes to individual items are not automatically reflected in the XAML unless modifications are made to the class contained in the collection. For example, if we update our `Egg` class to implement `INotifyPropertyChanged`, like this:

```
public class Egg: INotifyPropertyChanged
```

and modify the properties to call the `onChange` method, like this:

```
public decimal Price  
{  
    get => price;  
    set { price = value; onChange(); }  
}
```

Then when invoking a method in the VM class like this:

```
public void UpdateEgg()  
{  
    Eggs[0].Price = 100.00m;  
}
```

The binding in the XAML:

```
<Label Content="{Binding Eggs[0].Price}"></Label>
```

Will reflect the changes. Also notice that it is possible to reference a specific element of the collection in the XAML using the same syntax as in the C#. Unfortunately, it is not possible to change the zero to a variable directly. However, if your application needs it, then you could specify an index as a property in your VM class and use a converter to get the correct element from the collection.

One thing to be aware of is that assigning the elements of one collection of objects to another does not copy the object, but only a reference to the object. For example, if you write:

```
var eggs1 = new List<Egg>{new Egg()};  
var eggs2 = new List<Egg>();  
eggs2[0] = eggs1[0];
```

Then both eggs1 and eggs2 have a reference to the same object. Any changes you make to eggs1[0] will also show up in eggs2[0].

Also be aware that if you use populate the BindingList in the constructor in this way:

```
List<Egg> leggs = new List<Egg> { new Egg() };  
Eggs = new BindingList<Egg>(leggs);
```

Then any changes made to the Eggs BindingList<Egg> are also made to the leggs List<Egg>. This may be a desirable behavior, or not, depending on your use case.

ObservableCollection

The ObservableCollection class is very similar to the BindingList class with one main difference. ObservableCollection will not notify on changes to items in the collection even if you add the INotifyPropertyChanged interface to your class. In those cases where you do not need this behavior, then using this class would be an advantage in speed, since it does not require the overhead of listening for child events.

Foreach

When iterating over a collection using the for or while loop, a variable must be defined that does nothing but provide a current index. If that is the only reason for creating the variable, then it is better to use the foreach construct, like this:

```
foreach(var line in lines)
```

The loop variable (in this case, line) is directly populated with the next element of the collection until all elements have been selected, then the loop ends. In addition, this also saves writing

```
var line = lines[i];
```

inside the loop. In C# the default loop for collections should be the foreach. Only use the for and while loops if the loop counter variable is required for other purposes.

Break

If looping through an enumerable to search for an element and it is found, do not continue to loop through the remainder of the array. Instead, use the break statement to exit the loop early.

```
Animal foundAnimal = null;  
foreach(var animal in animals)  
{  
    if (animal.Age == 7)  
    {  
        foundAnimal = animal;  
        break; //this immediately exits this loop  
    }  
}
```

The break statement causes the execution to exit the innermost loop only. If this code were inside another loop, that outer loop would not terminate.

Exercise 9

Let us revisit the checkboard program. This time draw checkers on the board and place them according to the rules of checkers. To do that, we will use a collection to keep track of where to draw a checker and where not to. The drawing of a checker itself can be accomplished by creating an Ellipse object – with height and width the same - inside a Viewbox object and adding the Viewbox to each button as the Content property. Set the color of the Ellipse object as needed. Now that you begin to add business logic, use the MVVM pattern in your solution. When completed, compare your version with what is found in Exercise 9.

Reading and Parsing Files

Now that we know how to create and manipulate collections, we can finally bring data into our application from external sources. The simplest source of data to deal with is the file. A file can be unstructured text, binary image data or a structured text format such as CSV, JSON or XML. In any case, there are a few APIs available to read the data from a file. In any case, however, your code must make some assumptions about the contents of the file to be able to successfully process it into a useable form.

Most simply, we can use the analog of the method we used to write text to a file to read text from a file:

```
string text = File.ReadAllText("filename.txt");
```

This puts the entire contents of a text file into a single variable. If your data is known to have line breaks, like in the case of CSV formatted data where each line is a separate record, then you might take a shortcut and use either of these to get each line into a separate array or List element:

```
string[] lineArray = File.ReadAllLines("filename.txt");  
List<string> lines = (List<string>)File.ReadLines("filename.txt");
```

If the file contains binary data, like an image or a PDF, then you don't want to have the file processed as if it were containing text as this could lead to some characters being misinterpreted, so we can have C# read all the data in the file without any interpretation and return them to us in an array:

```
byte[] bytes = File.ReadAllBytes("filename.bmp");
```

A lower level approach to reading (and writing) files is to use the Stream set of classes. For example, we can read all the lines from a file like this:

```
List<string> text1 = new List<string>();  
FileStream stream1 = File.OpenRead("filename.txt");  
StreamReader r1 = new StreamReader(stream1);  
while(!r1.EndOfStream)  
    text1.Add(r1.ReadLine());
```

This is obviously more complex than the one line ReadLines() and, in fact, ReadLines uses a FileStream internally. So, there is not much reason to use the stream interface directly.

Assuming that we take the simplest approach with a text file:

```
string text = File.ReadAllText("filename.txt");
```

the variable text will contain all the characters from the file. If all we wanted to do is to display those characters in a TextBlock control, then we are done.

Typically, though, we need to parse the contents of the file to extract some ordered information. One of the most common file formats is CSV (comma separated values). If you create a .csv file, you will notice that Excel will claim it as a file type it knows and will populate a spreadsheet with the contents. For example, a csv file might look like:

```
Size, Color, Animal, Age  
large, green, dragon, 7
```

```
medium, orange, dog, 3
small, white, mouse, 1
```

Where each line contains several values, separated by commas, to form columns and rows of a table. The first line could be column names, as in this case, or that line is skipped and only data is present. If we use `ReadAllText()`, then the text variable will contain all this text and we'll need to parse the resulting data to extract the individual values and put them into our animal object:

```
public class Animal
{
    public string Size { get; set; }
    public string Color { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

The first step is to split the string into individual lines. We can use the `Split` method of the string class to do this:

```
string[] lines = text.Split(new char[] { '\r', '\n' },
    StringSplitOptions.RemoveEmptyEntries);
```

The `Split` method takes two parameters. The first parameter lists the characters to be used as the dividing characters between blocks of text. Here we are specifying our old friends, the line break characters. Each time `Split` encounters one of these characters, it creates a new entry in the output array and populates it with the characters from the beginning of the string or the last time a line break character was encountered until this character. Then it strips those characters from the beginning of the string. And populates a new entry in the variable `lines`.

The second parameter turns on the option to avoid creating empty entries in the array. This could happen to us if we had Windows style line breaks with both the `\r` and the `\n` characters. It would find the `\r`, create a new entry and populate it with the last word of the line, then it would find the `\n` character and, without this option turned on, would create a new, empty, entry – since there are no characters between the `\r` and the `\n`.

Once split on lines, we then need to split each line on the comma to get each item separately in an array and then populate the `Animal` object like so:

```
const int ANIMALINDEX_SIZE = 0;
const int ANIMALINDEX_COLOR = 1;
const int ANIMALINDEX_NAME = 2;
const int ANIMALINDEX_AGE = 3;
const int ANIMALELEMENTS_COUNT = 4;

List<Animal> animals = new List<Animal>();
foreach (var line in lines[1..])
{
    var element = line.Split(',');
    if (element.Length == ANIMALELEMENTS_COUNT &&
        int.TryParse(element[ANIMALINDEX_AGE].Trim(), out var age))
    {
        animals.Add(new Animal()
        {
```

```

        Size = element[ANIMALINDEX_SIZE].Trim(),
        Color = element[ANIMALINDEX_COLOR].Trim(),
        Name = element[ANIMALINDEX_NAME].Trim(),
        Age = age
    });
}
}

```

A few things to note. The `foreach` iterates over `lines[1..]`. This is a new syntax in C# version 8 and is only available if you are using .NET5 and above. It allows you to specify the range of array elements of interest. In this case, it is saying start at index 1 (just past the column titles line) and go to the end. So even though we do not want to loop over all elements, we can still use the `foreach`.

It is best practice to use constants for the index values. Because `split` is splitting only on the comma, we use the string method `Trim()` to ensure that any whitespace in between the commas and the words is removed. We validate the number of elements and ensure that the integer element is, in fact, an integer before attempting to populate a new `Animal` object and add it to the list of animals.

Of course, a full CSV file is more complex than what is presented here. For instance, it typically uses double quote characters to surround strings, so that embedded commas are not misinterpreted as column separators. Rather than implementing something more complex yourself, you should first check to see if .NET has an implementation or if one is available through NuGet. And, in fact, both options exist. There is a built-in `TextFieldParser` class and a NuGet package named `CsvHelper` (<https://joshclose.github.io/CsvHelper/>). I recommend you read through the documentation for both and see which of the two you would prefer to use. I suspect you will see that `CsvHelper` is a more comprehensive set of methods and a cleaner interface for populating objects. This is one of those rare times when the .NET class is inferior.

A more complex structure can be accomplished either through XML or JSON formatted text files. For these, it is not recommended to try and do it yourself. Instead, use the built-in classes.

For example, converting a C# object to JSON is as simple as:

```

var animals = new List<Animal>();
var jsonString = JsonSerializer.Serialize(animals);
File.WriteAllText(fileName, jsonString);

```

And reading it back can be accomplished by:

```

jsonString = File.ReadAllText(fileName);
animals = JsonSerializer.Deserialize<List<Animal>>(jsonString);

```

In both cases, the `Json` parser can deal with any arbitrarily complex object on the way in or out of the parser. Note that the `Json` parser will only serialize and deserialize public properties of the class. No variables, constants, or methods will be processed.

The XML methods are somewhat more complex because they require the use of a `Stream` object when serializing:

```

var serializer = new XmlSerializer(typeof(List<Animal>));
var fs = new FileStream(fileName, FileMode.Create);
var writer = new XmlTextWriter(fs, Encoding.Unicode);
serializer.Serialize(writer, animals);

```

```
writer.Close();
```

And, when deserializing:

```
var serializer = new XmlSerializer(typeof(List<Animal>));  
using Stream reader = new FileStream(fileName, FileMode.Open);  
animals = (List<Animal>)serializer.Deserialize(reader);
```

Comparing the two output files, you can see that the JSON is a significantly more compact expression of the same data. Because of that, and because JavaScript reads this format natively, it has become the current standard for data interchange between systems and on the web. There are, however, binary formats that are even more compact.

Sorting

When displaying a list of items to users, it is normal for the user to expect the list to be sorted. As well, a sorted list allows the programmer some flexibility in how the collection is stored internally. For that to work, the collection that underlies the List control must be sorted. .NET provides a Sort() function for lists and arrays. For lists of simple items or strings, use the built-in sort method without parameters is sufficient:

```
int[] intArray = new int[] { 6, 4, 2, 1, 3, 5 };  
Array.Sort(intArray); //rearranges the contents of intArray to 1...6  
  
var intList = new List<int>() { 6, 4, 2, 1, 3, 5 };  
intList.Sort(); //rearranges the contents of intList to 1...6
```

For complex objects, .NET cannot know how to sort since it is possible to sort on any combination of object properties in any order. Instead, we need to provide instructions for sorting. There are several approaches to providing these instructions: the Comparison method, the IComparable interface, and the IComparer interface.

The IComparable interface is implemented on the object to be sorted, like:

```
public class Animal: IComparable  
{  
    //properties, followed by  
    public int CompareTo(Animal a) => a.Size.CompareTo(Size);  
}
```

The IComparer interface is implemented on the object to be sorted, like:

```
public class Animal: IComparer<Animal>  
{  
    //properties, followed by  
    public int Compare(Animal x, Animal y) => x.Size.CompareTo(y.Size);  
}
```

The Comparison method does not require the class to implement an interface, instead, you provide a method to the Sort method, like so:

```
animals.Sort((a, b) => a.Size.CompareTo(b.Size));
```

This gives more flexibility than the interface approach because with the interface, we're hardcoding a specific sort item and order. Whereas with the Comparison method approach, we can pass in a different method to each Sort call, if desired. The downside of the Comparison method approach, though, is that the method is not part of the class and might be harder to find in a large project.

Notice that the Lambda syntax is used in all three cases. However, the last case extends the syntax a bit more because we are missing the function name and we do not include the data type of the variables. This is what is known as an anonymous method. Because we have created the method in place and only use it in this one place, there is no need to provide it with a name. As well, C# knows that the method required by `animals.Sort()` MUST provide two animal objects, so there is no need to explicitly say that.

Also notice that all comparison options return an integer value. The meaning of this returned value is: If > 0 , one is larger than the other, if < 0 , it is smaller, and 0 means they are equal. To reverse the order of the sort, change the sign of the comparison, like so:

```
animals.Sort((a, b) => -a.Size.CompareTo(b.Size));
```

More complex sorts are also possible by testing for equality on a comparison and, in that case, comparing a second property of the object. For example, if we want to sort by Size and by Age within Size, we do:

```
animals.Sort((a, b) =>
{
    var result = a.Size.CompareTo(b.Size);
    if (result == 0)
        result = a.Age.CompareTo(b.Age);
    return result;
});
```

And we can continue to add more comparisons until it is sorted exactly as desired.

Exercise 10

With the exercise, you will find a file named "Census2016 Canada.csv". It contains the total population of Canada from the 2016 census by age and by gender. Write an application to read and parse this file. Display the data in a Listbox, showing male, female, and total amounts. Allow the user to select the number of years in a group and display the totals for that group (that is, if they enter 5, then the groups should be 0-4, 5-9, 10-14...). Allow the user to sort the data by each column. Finally, show the total for each column (male, female, total) and the weighted average age. When completed, compare your version with what is found in Exercise 10.

Linq

When the C# language was developed, the prevailing programming philosophy was that object-oriented programming would solve the pressing development issues of the day and would lead to better and more bug free applications. This is, to a large extent, true. As programmers, we are more productive now that we can let the language and the framework worry about allocating and deallocating memory. We can find methods that act on a particular class more easily because they are associated with the class. However, bugs are not yet a thing of the past. Certainly, older style issues, like memory allocation issues are, for the most part, gone. But, because we can do more, we do. And this extra complication becomes harder to reason about and it may be impossible to determine the state of a complex application.

To solve this problem, functional programming has become more popular recently. With pure functional programs, it is possible to mathematically ‘prove’ the correctness of the application. There are many components that allow this, but one of them is immutability. That is, objects cannot be changed. Instead, a new object is created with the desired properties modified. I am sure you can see how this could be a problem for UI Controls. After all, we depend on being able to modify a property on an existing control to update user visible attributes. So, purely functional languages have some limitations in their use. However, the ideas have started to be adopted by other languages, including C#.

One of those functional programming ideas has resulted in the Linq library. Linq (or Language Integrated Query) is yet another way to manipulate lists and arrays. More precisely, it operates on classes that implement the IEnumerable interface. This interface is simple – it implements a method for advancing to the next element in the collection or returning to the start. Because it operates at such a simple level, the Linq methods can be used with all types of collections. Linq borrows from functional programming in that it does not modify the original collection. Instead, it creates a new, transformed collection. For example, the Sort method of a collection modifies the original collection to reflect a new order. Linq sorting creates a new collection with the new sort order but leaves the original collection untouched.

Linq also uses something Microsoft calls a ‘fluent’ interface. What they mean by this is that you can chain methods together in a single statement – making the code more readable. As well, when fluent and IEnumerable work together to delay execution until a final result is required, which keeps the efficiency high especially when using Linq to query a database.

If we used the previous techniques to filter and sort a list of records, we might write:

```
var eggs1e = new List<Egg>();
foreach (var egg in eggs1)
    if (egg.Price % 2 == 0)
        eggs1e.Add(egg);

eggs1e.Sort((a, b) =>
{
    var diff = (int)a.Size - (int)b.Size;
    return diff == 0 ? (int)a.Price - (int)b.Price : diff;
});
```

Which creates a new List of the eggs with even number prices and then sorts by price within each size. To write the same thing using Linq, you would write:

```
var eggs2e = eggs2.Where(e => e.Price % 2 == 0)
    .OrderBy(e => e.Size)
    .ThenBy(e => e.Price);
```

The Where method replaces the foreach loop. You can see the condition of the if statement in the anonymous method passed to the Where method. It expects a method that will take the collection's object type as its only parameter and return a boolean to indicate whether to include the item in the new list, or not. The OrderBy and ThenBy methods are also provided with anonymous methods. These are expected to return a simple data type that can be sorted by the built-in comparison mechanisms.

It is also possible to write multiple conditions into the Where clause, either by creating a compound test or by chaining multiple Where clauses. I find that the multiple clause approach is more readable, but that is really a personal and team preference.

Not only is the Linq approach more readable and takes less time to write, but it takes no more time to execute. Most importantly, modifying the logic to rearrange the sort conditions or update the direction (OrderByDesc) is trivial, compared to the standard approach. Remember, though, that even though a new collection object is created, it does not create new objects, so making a change to the eggs2[0].Size property may cause eggs2e to no longer be sorted.

Although Where will filter a collection and, if you filter appropriately, will return a single object, that object still ends up on a new IEnumerable that your code must reference into. If you know that only one item is required from the collection a better choice is to get that object directly by writing something like:

```
var animal = animals.FirstOrDefault(a => a.Age == 7);
```

First, FirstOrDefault, Single, and SingleOrDefault all return a single element. The differences between the four methods are that the First and FirstOrDefault methods will return the first match, while the Single and SingleOrDefault methods will only return an object if it is the *only* one to match. If more than one matches, an exception is thrown.

The FirstOrDefault and SingleOrDefault methods return null if a matching element cannot be found. The other two throw an exception if no matching element is found.

Besides filtering a list, we can convert a collection of one type of object into another type by using the Select method, like so:

```
var ages = animals.Select(a => a.Age).ToList();
```

In this case, the variable ages results in List<int> object. That is because we have extracted the integer Age from each Animal object in the animals collection using the Select method. Of course, more complex mappings are possible and new objects can be created from existing ones. If, for example, we define a new object:

```
public class SimpleEgg
{
    public decimal Price { get; set; }
    public bool IsHardBoiled { get; set; }
}
```

Then we can use select to map a collection of Egg onto a collection of SimpleEgg, like so:

```
var simpleEggs = eggs.Select(e => new SimpleEgg
{
    Price = e.Price,
    IsHardBoiled = e.IsHardBoiled
})
.ToList();
```

As the anonymous method passed to Select takes an Egg as an input parameter, and returns a SimpleEgg object. The compiler infers the datatype of the simpleEggs by what data type is returned by the anonymous method.

Finally, we can aggregate information using the Aggregate methods. This simplest of these is:

```
var totalAge = animals.Aggregate(0, (a, b) => a + b.Age);
```

Here the aggregate method uses the first parameter as a starting value (in this case, 0) and calls the provided anonymous method once for each element in the collection. The anonymous method receives the aggregation value so far (starting with the initial value) and the next element in the collection. It returns the result of the method's calculation that is then passed back into the function. In this way, the final, in this case, int value returned is the accumulation of all calculations done in the method.

Some aggregation methods are so common (Max, Min, Sum, Average) that custom methods are available for those operations without requiring the use of the Aggregate() method.

Another pair of useful Linq methods are Except and Intersect. They provide an easy way to determine the differences between two lists. For example,

```
var oldArray = new int[] { 1, 2, 3, 4, 5 };
var newArray = new int[] { 3, 4, 5, 6, 7 };

var added = newArray.Except(oldArray).ToArray(); //contains 6, 7
var removed = oldArray.Except(newArray).ToArray(); //contains 1, 2
var common = newArray.Intersect(oldArray).ToArray(); //contains 3, 4, 5
```

these three method calls give us information about the three things that could happen to a list: additions, deletions, and unchanged values.

Although this is interesting, what we normally want is to apply this not to arrays or lists of simple data types, but objects. To do that, we require a definition of what equality means for two objects. Fortunately, Except and Intersect both accept a second, optional, parameter that defines equality for the object of interest. To use it, we create a new class that implements the IEqualityComparer interface, like so:

```
internal class ItemEqualityComparer : IEqualityComparer<Item>
{
    public bool Equals(Item i1, Item i2)
        => i1?.Id == i2?.Id;

    public int GetHashCode(Item i) => i.Id.GetHashCode();
}
```

The interface requires that we implement the Equals method and the GetHashCode method. In this case, we are saying that if the Id properties of the two objects are the same, then they are the same. This is a common approach to take, especially if the data is coming from a database with auto assigned integer id values.

Create an instance of this class and pass it as the second parameter to the Intersect and Except methods.

Exercise 11

Repeat Exercise 10, but this time using Linq methods. When completed, compare your version with what is found in Exercise 11.

Classes in detail

It has been necessary to introduce the concept of classes to discuss other topics in this book. This should come as no surprise as classes are the fundamental building blocks of object-oriented programming languages, such as C#. As was said before, classes can be thought of as blueprints for a building. They are not the building itself but represent plans for the building. To create a building, it needs to be built and many can be built from the same set of blueprints. In the same way, a Class defines the properties and methods of an object, but it requires that we instantiate an object to create one. And, of course, many instances of the class can be instantiated from a single class definition.

We have already seen an example of this with how we populated our `List<Animal>` or `List<Egg>` or our `VM` class in the previous sections. First, we defined the class. Then, we use the 'new' keyword to create an instance of the class and populate it with values unique to that instance.

Inheritance

A class can inherit behaviors and properties of a base class. We have seen the ability of a class to do this from the very beginning of our journey. Our first class, `MainWindow`, that was created for us by the project creation tool, has a line that looks like:

```
public partial class MainWindow : Window
```

`MainWindow` inherits the behaviors and properties of the `Window` class. That is how we get `MainWindow` to have all those standard behaviors like resizing, a Title Bar, borders, and the minimize, maximize, and close widgets. `MainWindow` is called a derived class and `Window` is the base class. Notice the 'partial' keyword. With this keyword, we are indicating that the class named 'MainWindow' is defined across multiple files – the XAML file and its C# implementation being the other part. Visual Studio will stitch them together as it builds the project for us.

It is also possible to inherit from more than one class. You may want to do that if you have a class that needs to combine the behaviors of two distinct base classes. Classes that inherit from another class use the implementation written in the base class by default.

However, it is possible to override the implementation if the base class allows it. For example, `object` is the base class for all classes and it has a `ToString()` method defined that can be overridden like:

```
public override string ToString() => base.ToString();
```

In this case, the override does not do much as it overrides only to call the base class's `ToString()` implementation. However, it is possible to not use the base implementation at all or to use it and then do more. For example, overriding the `ToString()` method can look like:

```
public override string ToString() => $"{Size} | {Price} | {IsHardBoiled}";
```

Once you have done this, you will find that when debugging your code, Visual Studio will show this string instead of the object's name.

Interface

Classes can also implement an Interface. We have also seen that already as well:

```
public class Animal: IComparer<Animal>
```

Built-in interfaces are identified by the convention of starting the name with a capital 'I'. The difference between inheriting from a base class and implementing an interface is that interfaces proscribe the methods and properties that a class requires but do not actually provide code for them. As you had seen when adding the `IComparer` interface, our own custom implementation of the `Compare()` method was required.

As with inheritance, a class can implement multiple interfaces. This will be helpful if we want our object to implement both the `IComparer` and `INotifyPropertyChanged` interfaces. Just add a comma separated list of interfaces to implement after the colon.

Abstract Class

Sometimes you want to do more than just define the properties and methods that a class must have by using interfaces. Instead, you would like to provide a default implementation of the methods and properties. If that is the case, then an abstract class could be appropriate. This is like inheriting from a base class, except that it is not possible to create an instance of an abstract class. For instance:

```
var w = new Window();
```

is valid because `Window` is a regular class that we can inherit from. However, writing:

```
var b = new System.Drawing.Brush();
```

is not valid because `Brush` is an abstract class. To create a new brush, we must create a brush that inherits from `System.Drawing.Brush`, such as `SolidBrush` or `LinearGradientBrush`.

In our own code, we can use abstract classes to consolidate common code used across classes. For example, the `OnPropertyChanged()` method in each property of an `INotifyPropertyChanged` class is repeated everywhere, but if we wanted to write this code once and reuse it, we might write:

```
public abstract class BindingObject : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
        => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

Now instead of implementing the `INotifyPropertyChanged` interface in each class that requires it, we can inherit from `BindingObject` and have the implementation done for us. Doing it this way provides more consistency across our codebase and saves copy/paste into each class.

Notice the use the 'protected' keyword attached to the `OnPropertyChanged` method. The intention when specifying this is to limit access to this method to only those classes that inherit from this class. In other words, those that inherit from `BindingObject` will be able to access the method, but as far as the rest of the application is concerned, `OnPropertyChanged` is a private method.

The event, however, is public because we want other code in our project to access it.

Constructors

Sometimes, classes require initialization code to be executed. Again, we have already seen this with the `MainWindow` class. It has the following code:

```
public MainWindow()  
{  
    InitializeComponent();  
}
```

This is a special method with no return value and a name that matches the name of the class. This special method gets executed only once for each instance when that instance of the class is created with the 'new' keyword. This method is given the name 'constructor'.

Our `Animal` class did not have explicit code for a constructor, but we were still able to create an instance of the class using `new Animal()`. That is because there is, by default, a default do-nothing constructor implied.

In addition to explicitly coding a constructor like `MainWindow()` and having it do some work, we can also create constructors that require parameters. Recall that methods of the same name can have multiple versions each with a different parameter list. Constructors are no different. We can create as many constructors as needed, for example:

```
public Egg(decimal price)  
{  
    Price = price;  
}  
  
public Egg(bool isHardBoiled)  
{  
    IsHardBoiled = isHardBoiled;  
}
```

However, note that if a constructor with parameters is explicitly defined, the default no-parameter constructor is no longer implied. If still needed, it would need to be explicitly defined as well. With modern C#, these multiple constructors are no longer as common, since they can be replaced by using the default constructor and populating public properties directly:

```
var egg1 = new Egg {Price = 7m};  
var egg2 = new Egg {IsHardBoiled = true};
```

Rather than forcing us to figure out ahead of time what properties callers need initialized ahead of time, this approach leaves it open to the caller. There is one situation, however, that still may require multiple constructors, and that is where the thing to be initialized is not exposed as a public property, but that situation is not so common.

Singleton Class

There are times that it is necessary to protect access to some resource that we want to use in several places in our code. Examples of this include some piece of hardware like a serial port, a cash drawer, or a custom display. As well, we might want to put similar protection around a file, a database, or even our VM class.

For instance, to implement a simple logging mechanism to record user actions in our application, we will want to write to a single file, but from many locations in our code. If we instantiate an instance of our Log class, most likely the constructor would open a file for write access using the `FileStream` class and save the `FileStream` instance to use in any subsequent calls to write text to the log.

In this case, we would not want to use the `File.AppendAllText()` method because this would incur the overhead of opening and closing the file for every log entry. That is something we can do when occasionally writing to a file in the normal course of the application, but we do not want logging to affect the speed of our application.

If we created another instance of the Log class, the constructor would run again for this new instance and would attempt to open a new `FileStream` instance for writing to the same file. Unfortunately, Windows does not allow multiple writers of a file to exist at the same time, so the attempt would throw an exception. So, that means the code must share this one instance of the Log class to all the places in the code where logging is desired.

One way to do this is to instantiate the Log class in our `MainWindow` constructor and then pass this one instance of the Log class as an extra parameter to each method that wants to use it – likely all of them. This is painfully awkward, especially if we need to retrofit the Log class into an already existing application, since we would have to add a new parameter to every method in the application.

A much better way to deal with this need is using what is known as a singleton class, which would look like this:

```
internal class Singleton
{
    private static Singleton singleton;
    internal static Singleton Inst => singleton ?? (singleton = new Singleton());
    private Singleton(){}
}
```

By making the constructor private, we do not allow any code outside of this class to create an instance. Instead, we expose a static property 'Inst' and have it return either the already created Singleton stored in the variable `singleton` or create a new instance of `Singleton` and return that.

Notice the change in attribute to 'internal'. I use it here to show yet another accessibility level (besides public and private). The idea of internal is that only those inside the same project may access this class, property, or method. For small projects, this is effectively the same as public. However, if you are building a library to be used by others – via NuGet, for example – then separating public from internal allows you to identify those methods that are intended for external use from those to be used across classes within the same project.

Remember that the null-coalescing operator `??` returns the value of its left-hand operand if it is not null; otherwise, it evaluates the right-hand operand and returns its result. The `??` operator does not evaluate its right-hand operand if the left-hand operand evaluates to non-null. So, if this is the first invocation of `Inst` by the application, then the variable `singleton` will be null and the private constructor will be invoked and `singleton` populated with the instance. Every other time `Inst` is invoked, the already created `Singleton` object stored in `singleton` will be returned so every caller gets the same instance of the object.

Static Class

In addition to creating classes that reflect the system we are trying to model, we sometimes want to group together a set of static methods that perform similar functions. If the class contains nothing but static methods, we can declare the whole class to be static, like:

```
static class StaticClass
```

An example of this is the `System.Convert` built in class that contains a series of methods for converting one data type to another. If you attempt to write:

```
var c = new System.Convert();
```

you will get the error: *Cannot create an instance of the static class 'static class'.* You want this behavior if there are no instance specific properties in the class so your users do not accidentally create instances of classes for which there is no purpose, making it clearer to the users of your class how it is intended to be used.

Clone

There are times when it is convenient to make a copy of a class. For example, you may want to allow the user to modify a copy of an instance of a class instead of the original so that if they decide to cancel the operation, the copy can simply be discarded.

You could write code to create a new instance of the class and then copy the fields one at a time from the old to the new. However, a simpler approach is to use the built-in method available in all classes:

```
public InstanceClass Clone() => (InstanceClass)MemberwiseClone();
```

The limitation of the `MemberwiseClone()` method is that it only does a 'shallow' copy. That is, simple data types will be copied, but complex types will result in both objects having a reference to the same underlying object. But it is possible to deal with that issue by writing additional code, like:

```
public InstanceClass Clone()
{
    var i = (InstanceClass)MemberwiseClone();
    i.SubInstances = new List<SubInstance>();
    foreach (var si in SubInstances)
        i.SubInstances.Add(si.Clone());
    return i;
}
```

Here we know that our instance class contains a list of `SubInstances`, so we clear out the list after `MemberwiseClone()` has put in a reference to the original list and then populate the new list with clones of all the `SubInstance` objects. Of course, that implies that `SubInstance` also implements the `Clone()` method and potentially repeats this process for all of its complex data types.

Override

In addition to being able to override the `ToString()` implementation in a class, the base object provides two other methods that can be overridden: `Equals()` and `GetHashCode()`. We generally do not need to override these because the default implementation is useful in most situations. However, there are times when it would be useful to redefine what it means for one instance to equal another instance of the same object. This is especially true if you use cloning to make copies since the default

implementation checks the instance's reference - basically, its memory location – and only declares two objects equal if both references are to the same object.

To circumvent this, we typically want to check some subset of the instance's properties against the other instance to see if they are the same and report equality if they are. If we attach a unique identifier to the object, then only one field needs to be compared. For instance, we could assign a unique integer value to each object and store it in a property, like:

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

then,

```
public override bool Equals(object obj) => (obj as InstanceClass)?.Id == Id;
```

to report equality if the Id property of the instance provided as a parameter is identical to the Id property of the current object.

However, if you override only Equals, you will receive a warning *'class' overrides Object.Equals(object o) but does not override Object.GetHashCode()* as these must be overridden together. The value that GetHashCode() generates is used to uniquely identify an object and is used when deciding whether a Dictionary already contains the object or not and for List.Remove() to determine the matching item to remove. Since a different result from comparing hash codes or using the Equals() method could result in unexpected behavior, we really do want those two methods to agree. Fortunately, generating a custom hash code for our object is straightforward, as in:

```
public override int GetHashCode() => Id.GetHashCode();
```

where the hashcode for our object is just the hashcode for the one field that we are comparing in the Equals() method. Actually, the hashcode for an int data type is simply the int value with no transformation applied, so you could just return Id.

If, however, it is necessary to compare multiple properties, then newer versions of .NET provide this mechanism:

```
public int GetHashCode(InstanceClass i) => GetHashCode.Combine(i.Id, i.Other);
```

where up to eight objects can be passed and a hash value returned. It is also possible to use the tuple construct to do the same, as in:

```
public int GetHashCode(InstanceClass i) => (i.Id, i.Other).GetHashCode();
```

We can also create abstract classes with methods that can be overridden by using the 'virtual' keyword in the base class. In this case, the default implementation is still provided, but the derived class may also choose to implement its own version or do some custom work before turning over the job to the base class's implementation.

Extension methods

There are times when you would like an existing class to implement a new method but it doesn't seem like it would be worthwhile to create a derived class either because the class you want to add to is so

commonly used that no one will remember to use your class or because you want this to apply to a base class and be accessible to all of its derived classes. C# offers extension methods for this purpose.

You have already used extension methods when you use Linq. All the Linq methods are implemented as extensions to the existing `IEnumerable<T>` class and its derived classes. You can see the value, I think, it being able to add Linq to your code without having to update your `List<T>` declarations to be `List2<T>` to be able to access linq.

To do this, they declare their methods without specifying the exact data type of the object in the list. So, for example, you can use the Linq Where method on `List<int>`, `List<string>`, and `List<some other object>` equivalently. They could have declared the supported type as `IEnumerable<object>`, but that causes difficulties because `object` is no longer strongly typed and extra casting is required. Instead, it is declared with a placeholder that the compiler will substitute as needed. For instance, the Where extension method can be declared this way:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> f)
{
    foreach (T item in source)
    {
        if (f(item))
            yield return item;
    }
}
```

With `T` as the placeholder for the type. So, the return value is an `IEnumerable` of type `T`, it applies to objects of type `IEnumerable<T>` (ie. Any `IEnumerable`), it expects a method that passes a type `T` and returns a boolean. The `foreach` loop in the code iterates over all items in the `IEnumerable` and, if the method `f` indicates there is a match, then it adds the item to the list. In fact, the Where could be written as

```
IEnumerable<Egg> eggs2f = eggs2.Where<Egg>((Egg e) => e.Price % 2 == 0);
```

And explicitly state the type of corresponding to each of the `T` in the definition, but the compiler is smart enough to figure out that if `eggs2` is a `List<Egg>`, then `T` must be `Egg` in this case and substitutes for you.

One other bit of new syntax is the 'yield' keyword. What this indicates to the compiler is to create an unnamed `IEnumerable<T>` to hold the results and put each item where yield is indicated into that collection. It is simply saving us from having to name and create a local collection.

You can also add your own custom extension methods with this syntax:

```
internal static int GetNameLength(this InstanceClass i) => i.Name.Length;
```

This is obviously a contrived example, but the important thing to notice is the use of the 'this' keyword. With this code added to a static class, any `InstanceClass` instance will now have a `GetNameLength()` method that, in this case, takes no parameters. You can, of course, add parameters if required and make the body of the method as complex as necessary.

Navigating Multi-Window Applications

To this point, the applications we have developed have only required one window. Generally, though, normal applications require multiple windows. A common pattern is known as List/Details. In this pattern, one window of your application shows a list of whatever set of objects your application manipulates: invoices, emails, contacts, or playlists. Usually, the items in this list give enough detail to distinguish one row in the list from another, but no more. Clicking on a row in the list causes a second window to appear showing the full detail of the selected item. To add an item to the list or edit an item on the list, a third window could appear containing an editable version of the item. This will look different depending on the platform – on a phone you will only see one of these windows at a time, for example – but the general pattern does not change.

To accomplish this behavior in WPF, we can put the list part of our application in MainWindow and then add a second Window class to show the detail part of our application. To do this, first click Project -> Add Window. When the dialog appears, update the name, and click the Add button. This generates a new class that inherits from Window. It will have its own .xaml and .xaml.cs files where we can handle events specific to that window.

Now that we have a class, we need to make the window appear to the user on demand. Because the Window class is just another class, we can create an instance of it in the same way as we have created all other instances of classes – with the ‘new’ keyword, like so:

```
DetailsWindow dw = new DetailsWindow();
```

Now a new instance of the details window exists, but this is not yet enough to show it. To do that we need to invoke a Window method:

```
dw.ShowDialog();
```

We did not need to invoke this, or any other, method to show our MainWindow class because that code is autogenerated by Visual Studio. However, it does exist. In fact, you could create a new instance of MainWindow in the same way. If we add these two lines to an event handler in MainWindow(), we can cause DetailsWindow to appear on a user action.

If we cause the new window to be displayed to the user with the ShowDialog() method, the new window will take over all input for the application. Until the new window is dismissed, MainWindow cannot be interacted with. This is known as a modal dialog. This may be useful in some situations, like when your application cannot continue without some user input. Other times, however, you may still want to interact with the list portion of your application to, for instance, show a second detail window beside the first.

To do this, we can invoke a different approach to displaying the window, like this:

```
dw.Show();
```

With Show(), MainWindow is still active and can be interacted with. It means that the event handler that created and displayed DetailsWindow is still active and can be clicked multiple times. Each click will create a new instance of the DetailsWindow class. Creating an infinite number of windows can be fun, but normally we want to limit the number of DetailsWindow instances that can be shown in some way.

In the simplest case, we may want to limit to one instance of `DetailWindow`, but still interact with `MainWindow` to also cause an `EditWindow` instance to be simultaneously active. To limit the number of instances of `DetailWindow`, we need to modify our code.

First, move the definition of the variable `dw` out of the event handler method and make its scope global to the `MainWindow` class. When the variable `dw` was local to the event handler method, we lost access to the `DetailWindow` instance and can no longer call methods of that object. Note that even though our reference to that `Window` was lost, the window did not disappear – that is because the application still holds a reference to the window and the garbage collector will not remove it. By moving the variable to the global scope within the `MainWindow` class, we retain a reference to the window past the end of the event handler method.

Next, we need to create an instance of `DetailsWindow` only if we have not created one before, like so:

```
DetailsWindow dw;
void displayWindow()
{
    if (dw == null)
    {
        dw = new DetailsWindow();
        dw.Show();
    }
}
```

Basically, if the `dw` variable is null, we have not yet created a `DetailsWindow` instance. However, once created, `dw` is no longer null and we know it still exists.

However, we are not done yet. When the user closes the details window by clicking the top right corner 'close' button, our code is not aware of the occurrence of that event. Because it continues to hang onto the instance of `DetailsWindow`, `dw` has no way of getting set back to null. Because of our check, we end up having only one instance of `DetailsWindow` to appear until we restart the application.

To fix this, we need to listen for the window close event on `DetailsWindow` and act on it. We have used XAML to register for events up until now because the events were occurring in the same window class as the event handler. Now, however, we want to listen for events in a different class. To do that, we only need to add a couple of lines of code, like so:

```
DetailsWindow dw;
void displayWindow()
{
    if (dw == null)
    {
        dw = new DetailsWindow();
        dw.Closed += DetailsWindow_Closed;
        dw.Show();
    }
}

private void DetailsWindow_Closed(object sender, EventArgs e) => dw = null;
```

To register for the `Closed` event, we add – using the `+=` syntax – a method that we want to add to the list of methods to be called when the `Closed` event occurs. That is what XAML has been doing for us all

along with our button click and other event handlers. Notice that we provide the method name without brackets on the right side of the assignment statement. Methods themselves are variables – as we have seen when passing anonymous methods as parameters to Linq methods. The parameter list of `DetailsWindow_Closed` should also look familiar. It is the same two parameters as every other event handler we have ever written or had written for us.

`DetailsWindow_Closed` is not particularly complex. All it does is sets the variable `dw` back to null. Now the next time `displayWindow()` is called after the window is closed, `dw` will be null and a new `DetailsWindow` instance will be created.

We could also listen for other events on the `DetailsWindow` instance. For example, we could listen for the `Closing` event. This even occurs before the window is actually closed, so it is possible to stop the user from closing the window by setting `e.Cancel` to true.

The `DetailsWindow` typically displays information about one of the items we are working with. Somehow, we need to inform `DetailsWindow` about which item to display. There are several methods for this:

We could modify the constructor of `DetailsWindow`:

```
Animal animal;
public DetailsWindow(Animal animal)
{
    InitializeComponent();
    this.animal = animal;
}
```

And pass the item:

```
dw = new DetailsWindow(selectedAnimal);
```

Or we could add a public property to `DetailsWindow`

```
public Animal Animal { get; set; }
public DetailsWindow()
{
    InitializeComponent();
}
```

And set that property:

```
dw = new DetailsWindow
{
    Animal = selectedAnimal
};
```

With these two approaches, the object being passed is divorced from the `ViewModel` class. That could be a good thing if the UI is required to allow for the canceling of changes. If the code is making changes to a copy of the object, then the cancel feature can be implemented simply by deleting the copy.

However, if the requirement is rather to reflect changes immediately, then we could instead pass the entire `ViewModel` class and define a property – probably called `SelectedAnimal` in this case – bound to the `SelectedItem` attribute of our `ListBox` control. Because both windows are bound to the same `vm`

class, then any changes made in one window will be reflected immediately in the other window as windows will be listening for changes in the VM object. If we make the ViewModel class a singleton class, then each window that requires access to it can simply access the object through the Instance method and then no parameters are required at all.

The implementation of the DetailsWindow display so far is limited in that we are only tracking the opening and closing of one details window. If, though, the app requirement is for multiple details windows to be open simultaneously, then we need to upgrade our handling of windows.

Specifically, instead of a single variable to track if the window is displayed or not, we need a collection of variables. Fortunately, we have already discussed various collection objects. In this case, we have a pair of objects that should be tracked together: the window instance and the item to be displayed. The collection object that best reflects this is the Dictionary. But in the Dictionary, one object needs to be the key and the other the value. Since the item is always present and the window may be there and may not, then it makes sense to make the item the key and the window as the value, like this:

```
Dictionary<Animal, DetailsWindow> windows = new Dictionary<Animal, DetailsWindow>();
void displayWindow()
{
    if (!windows.ContainsKey(selectedAnimal))
    {
        var dw = new DetailsWindow
        {
            Animal = selectedAnimal
        };
        windows.Add(selectedAnimal, dw);
        dw.Closed += detailsWindow_Closed;
        dw.Show();
    }
}
private void detailsWindow_Closed(object sender, EventArgs e) =>
    windows.Remove((sender as DetailsWindow)?.Animal);
```

The main two differences are that a) we are using ContainsKey() to determine whether the selected animal already has a details window associated and b) we are using Add/Remove to manipulate the dictionary. Note that the variable sender is the instance of DetailsWindow that is closing, so in this case its Animal property can be used as the key to the dictionary.

Another note about ContainsKey and Remove when used with complex objects. In this case, two objects are equal if they are, in fact, the same object. If they are different instances with the same values, they are not considered equal. To change this behavior, we would need to override the Equals() and GetHashCode() methods, like so:

```
public override bool Equals(object o) => (o as Animal)?.Name == Name;

public override int GetHashCode() => Name?.GetHashCode() ?? 0;
```

In this case, we are saying that two Animal instances are the same if their Name properties are the same. We also should make their hash codes (used by Dictionary to store instances in its data structure for fast retrieval) compare only on the name for consistency. The Name? means that it will only try to execute GetHashCode if the contents are not null. If it is null, then the ?? will cause the entire expression to resolve to zero and that will be returned as the int hash code.

Displaying tables of data

DataGrid

A DataGrid is yet another way to display a list of items. As the name implies, it is a grid (like an Excel spreadsheet) and is most used when needing a quick way of connecting from a database table, as it has built in handling for that. As well, cells in a DataGrid can be of many types, including checkboxes, buttons, comboboxes, and of course textboxes.

The DataGrid control also allows the user to edit and add rows, resize columns, and sort by column by clicking on the column header. Editing a row means that the underlying object of that row is being modified by the DataGrid. Adding a row creates a new object and adds it to the collection bound to the row.

Imagine an application that is required to display this class:

```
public class Sample
{
    public bool CheckBoxData { get; set; }
    public string TextData { get; set; }
    public ComboBoxOptions ComboBoxData { get; set; }
    public DateTime DateData { get; set; }
}
```

Where:

```
public enum ComboBoxOptions
{
    Option1,
    Option2,
    Option3
}
```

in a tabular form. We could have a ViewModel class that exposes the list of these objects through the ObservableCollection class like this:

```
public ObservableCollection<Sample> Samples { get; set; } = new ObservableCollection<Sample>()
{
    new Sample {CheckBoxData = true, ComboBoxData = ComboBoxOptions.Option1,
                DateData = DateTime.Now, TextData = "This is a sample"},
    new Sample {CheckBoxData = true, ComboBoxData = ComboBoxOptions.Option2,
                DateData = DateTime.Now.AddDays(1), TextData = "This is a sample too"},
};
```

Then, using the DataGrid control, we could write something in the XAML like this:

```
<DataGrid ItemsSource="{Binding Samples}" AutoGenerateColumns="False"
    Style="{DynamicResource DataGridStyle}"
    CellStyle="{StaticResource CenterItemsStyle}"
    AlternatingRowBackground="LightGray" GridLinesVisibility="None"
    CanUserAddRows="False" CanUserDeleteRows="False" CanUserResizeRows="False">
    <DataGrid.Columns>
        <DataGridTextColumn Width="*" Header="Text Column" Binding="{Binding TextData}"/>
        <DataGridCheckBoxColumn Width="*" Header="Check" Binding="{Binding CheckBoxData}"/>
        <DataGridComboBoxColumn Width="*" Header="Combobox"
            ItemsSource="{DynamicResource OptionSource}"
            SelectedValueBinding="{Binding ComboBoxData,
                UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}"
```



```

                SelectedValuePath="Id"
                DisplayMemberPath="Description"/>
        <DataGridTemplateColumn Width="*" Header="Custom Column"
                CellTemplate="{StaticResource DateTemplate}"
                CellEditingTemplate="{StaticResource EditingDateTemplate}" />
    </DataGrid.Columns>
</DataGrid>

```

Notice that a DataGrid uses the same ItemsSource attribute as the ListBox to bind to the Samples ObservableCollection object. If we do not set AutoGenerateColumns to false, the DataGrid will automatically create columns for all the public properties of the class. This may be useful in limited situations, but generally we want to avoid this because there will usually be properties not suitable for display to the user.

The DataGrid in this instance is limited by disabling the adding of rows (CanUserAddRows), the deleting of rows (CanUserDeleteRows) and resizing rows (CanUserResizeRows). Enabling those gives the user the ability to add and subtract from the ObservableCollection. However, I find that users prefer to be provided with buttons and a separate dialog to accomplish the insertion. As always, each use case is different, so it may be worth exploring whether the users of your application will be comfortable with in table editing.

As far as styling, we can specify the style of the DataGrid control and the style of cells using a resource for each. For the DataGrid as a whole, we can customize multiple styles for things like drag and drop, focus, and the rows. Typically, you will want to override the default style for the column headers. In this case, we create a style resource for the column header and then apply that style via another style resource. Or you could apply the ColumnHeaderStyle directly in the body of the XAML.

```

<Style x:Key="DataGridStyle" TargetType="{x:Type DataGrid}">
    <Setter Property="ColumnHeaderStyle" Value="{DynamicResource ColumnHeaderStyle}"/>
</Style>
<Style x:Key="ColumnHeaderStyle" TargetType="DataGridColumnHeader">
    <Setter Property="Height" Value="30"/>
    <Setter Property="Background" Value="DarkGray"/>
    <Setter Property="Foreground" Value="GhostWhite"/>
    <Setter Property="HorizontalContentAlignment" Value="Center"/>
    <Setter Property="FontSize" Value="18" />
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="ToolTip" Value="Click to sort."/>
        </Trigger>
    </Style.Triggers>
</Style>

```

To style a cell of the DataGrid, you could specify something like this:

```

<Style x:Key="CenterItemsStyle" TargetType="{x:Type DataGridCell}">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type DataGridCell}">
                <Grid Background="{TemplateBinding Background}">
                    <ContentPresenter VerticalAlignment="Center" />
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>

```

```
</Setter>
</Style>
```

In this case we set the cell to display its contents centered vertically in the cell. The default is to align to the top of the cell. When looking at the XAML required to set this particular style, it is clear that the DataGrid control is a complex beast.

The columns of the DataGrid control in this example show the types of columns available: text, checkbox, combobox, and custom. As well as using the combobox column type, it is possible to create a custom column to implement the combobox. This approach can be found in many online tutorials because the binding can be much simpler.

The text and checkbox columns are simple to implement:

```
<DataGridTextColumn Width="*" Header="Text Column" Binding="{Binding TextData}"/>
<DataGridCheckBoxColumn Width="*" Header="Check" Binding="{Binding CheckBoxData}"/>
```

Where it is possible to easily set the basics: width, header text, and binding.

The combobox column, though, has a unique complexity because it needs to bind to both the current selection for the row and to the list of possible options.

```
<DataGridComboBoxColumn Width="*" Header="Combobox"
    ItemsSource="{DynamicResource OptionSource}"
    SelectedValueBinding="{Binding ComboBoxData,
        UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}"
    SelectedValuePath="Id"
    DisplayMemberPath="Description"/>
```

The ItemsSource attribute identifies the source of the list of options. To create this source requires a couple of steps. First, we need to define a class that will map our options enum to a string:

```
public class Option
{
    public ComboBoxOptions Id { get; set; }
    public string Description { get; set; }
}
```

Then create a collection class that we can bind to:

```
public class OptionCollection : ObservableCollection<Option>
{
}
```

Then, we need to use that class to create a resource, like this in the file App.xaml:

```
<Application.Resources>
    <local:OptionCollection x:Key="OptionSource">
    </local:OptionCollection>
</Application.Resources>
```

We could populate it here, but typically we will want to add elements to it dynamically so it can be changed as required during the run of the application, by doing this in the constructor for the MainWindow class:

```
var oc = Application.Current.Resources["OptionSource"] as OptionCollection;
oc.Add(new Option { Id = ComboBoxOptions.Option1, Description = "Option One" });
oc.Add(new Option { Id = ComboBoxOptions.Option2, Description = "Option Two" });
oc.Add(new Option { Id = ComboBoxOptions.Option3, Description = "Option Three" });
Application.Current.Resources["OptionSource"] = oc;
```

Which takes each of the options from the enum and associates a description to it. Now, the combobox will display one of the three descriptions – or be empty.

The `SelectedValueBinding` attribute is the one to use to bind the current row's selection to the cell. In this case, we are triggering on `PropertyChanged` because it is not guaranteed that the user will click elsewhere and cause the cell to lose focus before exiting – which would cause lost modifications. This is the safer approach.

The `SelectedValuePath` is not a binding but instead the name of the property the combobox should use from the `ItemsSource` objects to match the `SelectedValueBinding` value to the `ItemsSource` value. The `DisplayMemberPath` is also not a binding, but the name of the property the combobox should use to find the displayable element from the `ItemsSource` objects.

If setting up a combobox in this way seems like a daunting task, then it may be that using a custom column might be easier. In the example:

```
<DataGridTemplateColumn Width="*" Header="Custom Column"
    CellTemplate="{StaticResource DateTemplate}"
    CellEditingTemplate="{StaticResource EditingDateTemplate}" />
```

We specify two resources – one for display and one for editing – to define the contents of a Template, or custom, column. If no editing is required, then the editing template is not required. The example displays a date in an interesting way, but when editing, reverts to a standard `DatePicker` control, like this:

```
<DataTemplate x:Key="DateTemplate">
    <Grid Width="50" Height="35">
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Border Grid.Row="0" Margin="0,5,0,0"
            Background="LightGray" BorderBrush="Black" BorderThickness="1">
            <TextBlock Text="{Binding DateData, StringFormat={}{0:MMM}}"
                FontSize="8" HorizontalAlignment="Center"/>
        </Border>
        <Border Grid.Row="1" Margin="0,0,0,5"
            Background="LightGray" BorderBrush="Black" BorderThickness="1">
            <TextBlock Text="{Binding DateData, StringFormat={}{0:yyyy}}"
                FontSize="8" FontWeight="Bold" HorizontalAlignment="Center"/>
        </Border>
    </Grid>
</DataTemplate>
<DataTemplate x:Key="EditingDateTemplate">
    <DatePicker SelectedDate="{Binding DateData}"
        VerticalContentAlignment="Center" Height="35" />
</DataTemplate>
```

All of this should be familiar, with a DataTemplate specifying a Grid or a DatePicker binding to an element from our Sample class. What is different, though, is that the bindings are being specified in a resource. This could also have been written directly in line like:

```
<DataGridTemplateColumn Width="*" Header="Custom Column"
                        CellTemplate="{StaticResource DateTemplate}">
    <DataGridTemplateColumn.CellEditingTemplate>
        <DataTemplate>
            <DatePicker SelectedDate="{Binding DateData}"
                        VerticalContentAlignment="Center" Height="35" />
        </DataTemplate>
    </DataGridTemplateColumn.CellEditingTemplate>
</DataGridTemplateColumn>
```

But once things get complex, can cause the reader to lose track of the forest for the trees.

When using this approach, it is possible to directly bind to any property from the ViewModel class, making the combobox bindings much easier to accomplish.

ListView

The ListView control inherits from ListBox and adds the concept of a View. For example, the Windows File Explorer has Details, List, and Icon views of the files in a directory. By changing the View property in code, it is possible to switch between the various views. However, it is not necessary to use the View property at all. If not used, then a ListView and a ListBox are identical.

To create something like the DataGrid above, you could write:

```
<ListView Margin="5"
          ItemsSource="{Binding Samples}"
          Template="{StaticResource ListViewHeader}"
          ItemTemplate="{StaticResource ListViewItem}"
          HorizontalContentAlignment="Stretch">
</ListView>
```

Where we use the Template attribute to define the column headers and the ItemTemplate to define the look of each of the rows:

```
<ControlTemplate x:Key="ListViewHeader">
    <DockPanel LastChildFill="True">
        <Grid DockPanel.Dock="Top" Height="30">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Label Grid.Column="0" Style="{StaticResource LVHeaderStyle}" Content="Text" />
            <Label Grid.Column="1" Style="{StaticResource LVHeaderStyle}" Content="Checkbox" />
            <Label Grid.Column="2" Style="{StaticResource LVHeaderStyle}" Content="Combobox" />
            <Label Grid.Column="3" Style="{StaticResource LVHeaderStyle}" Content="Custom" />
        </Grid>
        <ItemsPresenter></ItemsPresenter>
    </DockPanel>
</ControlTemplate>
```

Where the LVHeaderStyle resource is a duplicate of the ColumnHeaderStyle resource for the DataGrid. To get the columns to appear at the top, we use a DockPanel to pin a fixed size Grid control to the top of the area and fill it with the columns. The ItemsPresenter control fills the rest of the area and is a placeholder for the ListView to place the rows of data. To handle clicking to sort, replace the Label controls with Button controls and handle the click events as you would normally.

To define the look of each row, use this resource:

```
<DataTemplate x:Key="ListViewItem">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <TextBox Grid.Column="0" Text="{Binding TextData}"
      VerticalContentAlignment="Center" BorderThickness="0" />
    <CheckBox Grid.Column="1" IsChecked="{Binding CheckBoxData}"
      VerticalAlignment="Center" HorizontalAlignment="Center" />
    <ComboBox Grid.Column="2"
      VerticalContentAlignment="Center"
      ItemsSource="{Binding DataContext.Options,
        RelativeSource={RelativeSource FindAncestor, AncestorType=ListView}}"
      DisplayMemberPath="Description" SelectedValuePath="Id"
      SelectedValue="{Binding ComboBoxData}" />
    <DatePicker Grid.Column="3" SelectedDate="{Binding DateData}"
      VerticalContentAlignment="Center" Height="35" BorderThickness="0"
      Margin="25,0,0,0" />
  </Grid>
</DataTemplate>
```

Again, we are specifying column widths inside a grid and then placing the various controls in that grid. In this case, the VerticalContentAlignment property is set for each separately – although the CheckBox requires that the VerticalAlignment property is set. To remove all the borders, the TextBox and DatePicker controls have their BorderThickness properties set to zero.

This should all look familiar to you, other than the ComboBox control's ItemSource property definition. Again, the ComboBox has two things it needs to bind with – the current selection for that row and to the list of possible selections. This time, the list of options is defined directly in the ViewModel class with this code:

```
public BindingList<Option> Options { get; set; } = new()
{
    new Option { Id = ComboBoxOptions.Option1, Description = "Option One" },
    new Option { Id = ComboBoxOptions.Option2, Description = "Option Two" },
    new Option { Id = ComboBoxOptions.Option3, Description = "Option Three" }
};
```

But we cannot access this property directly because the ComboBox's data context is the specific Sample object for the current row being displayed. To get at the Options object, we need to break out of this context and bubble up to the DataContext as assigned in the C# code. That is what RelativeSource does.

It allows us to specify that we want, in this case, the DataContext associated with the ListView that is the parent of this ComboBox control. Then, in that context, we can grab the Options object and bind to it.

If you did want to define a View for a ListView we could define a GridView, like this:

```
<ListView ItemsSource="{Binding Details}" SelectedItem="{Binding Detail}"
    HorizontalContentAlignment="Stretch" HorizontalAlignment="Stretch" Name="lvDetails">
    <ListView.View>
        <GridView>
            <GridViewColumn Header="Product" DisplayMemberBinding="{Binding ProductId}"/>
            <GridViewColumn Header="Qty">
                <GridViewColumn.CellTemplate>
                    <DataTemplate>
                        <TextBlock Text="{Binding Quantity}" TextAlignment="Right"/>
                    </DataTemplate>
                </GridViewColumn.CellTemplate>
            </GridViewColumn>
        </GridView>
    </ListView.View>
</ListView>
```

In this case, we are defining a GridViewColumn with a Header attribute to create columns with headers. In the simple case, we can provide a DisplayMemberBinding attribute for binding to the data we want to show as text. In a more complex case, we need to define a DataTemplate where we define the structure of the contents of the column.

The one annoying issue with this approach is that, if we do not specify a fixed column width, the ListView will size each column to fit the longest text currently in that column. If the table is to be used only to display unchanging data, then this is not a problem. But, if the data changes and new entries are longer, the columns do not resize to accommodate. Of course, the user can always resize the columns manually with a GridView – something that cannot be done with the previous ListView approach – but forcing the user to constantly resize columns can get annoying quickly. Fortunately, there is a solution, but it does take a few steps. First, we need to register for PropertyChanged events in our xaml.cs file. We have done this before when we monitored the Close event in our child window and the syntax is the same, like:

```
vm.PropertyChanged += vm_PropertyChanged;
```

and then we need to define the vm_PropertyChanged method like so:

```
private void vm_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName == "Details")
    {
        foreach(var c in (lvDetails.View as GridView).Columns)
        {
            if (double.IsNaN(c.Width))
            {
                c.Width = c.ActualWidth;
                c.Width = double.NaN;
            }
        }
    }
}
```

The idea is that whenever our viewmodel class puts out a notification that some property has changed, we will be notified and our `vm_PropertyChanged` method will be called. The event arguments parameter tells us the name of the property that has changed. If it is a property that contains information that could change the width of the text in a column, then we need to resize the columns. Since the 'auto' width setting is represented in code as the `double.NaN` value, then we know which columns are set to auto. If the current column is an 'auto' column, change its value to be `ActualWidth` – which contains the length of the longest item in that column – and then change it back to auto. This has the effect of resizing the columns for the new data.

Although the `ListView` versions display the same data as the `DataGrid` version, there are some differences. The main difference is that the `DataGrid` has a display view and an edit view and can show separate visuals for each. The `ListView`, at least as constituted here, only has one view and that is the edit view for the active controls. On the other hand, getting into edit mode on the `DataGrid` can be non-intuitive for the user. To change the `ComboBox` value on the `DataGrid`, the user must click once to select the row, a second time to get into edit mode, a third time to get the dropdown to display, and a fourth time to select a new option. With the `ListView`, clicking on the `ComboBox` of any row immediately opens it, requiring only one more click to select a new option, for a total of only two clicks. The other controls also require extra clicks to cause an edit. This can make using the `DataGrid` frustrating for users.

Exercise 12

Using the 'Sample' class defined above, implement an application that shows and edits the `Sample` type data in a tabular form using the `DataGrid`, `ListView` and `ListBox` controls. Use one window and have buttons to switch between the different table controls. When completed, compare your version with what is found in Exercise 12.

Accessing databases with SQL

A file can be a perfectly reasonable way to store unstructured data, or a small amount of structured data via JSON, CSV, or XML formatted files. But if the complexity of the data grows or if modifying and adding to the data is required, then a database is a much better mechanism for long term storage of structured data, as files are most easily appended to or completely overwritten.

Microsoft offers a free version of their database application, SQL Server, to install on your own Windows computer. Go here: <https://www.microsoft.com/en-us/sql-server/sql-server-downloads> and click the Download Now button below the Express special edition section. Run the installer and select the Basic installation option. At the end of the installation process, there will be a button to 'Install SSMS'. Click that and follow through with the installation of Sql Server Management Studio (SSMS). SSMS provides a Visual Studio style interface to create SQL queries and view results. This process also installs Azure Data Studio, a newer application that provides a Visual Studio Code style interface for manipulating databases.

So that we have a database to work with, go here: <https://github.com/microsoft/sql-server-samples/blob/master/samples/databases/northwind-pubs/instnwnd.sql>. Click the download button. Right-click on the text, select Save As... and save on your local machine as instnwnd.sql.

Once installed, run SSMS (in Windows 10, the quickest way is to type ssms in the search bar. The app will show up as first on the list). The first dialog (Connect to Server) should contain all the correct values by default. Click the Connect button.

Click File -> Open -> File... and find the instnwnd.sql file saved from above. Open it. Click the Execute toolbar item. You should now have an instance of the Microsoft NorthWind sample database to work with. To verify that the database has been created correctly, change the database selection to 'Northwind' (the default will be 'Master') and do a simple query with SSMS, like:

```
select * from customers
```

and verify that the tool displays 90+ records with customer details.

Now that we have a database to work with, we can start writing some code to access the database. There are 5 basic operations we would like to do from C# -- connect to the database, read records, insert new records, update existing records, and delete existing records. An application whose primary purpose is to do these actions is referred to as a CRUD application. That is not a comment on the app or your talent as a programmer, but an acronym for an application that performs Create, Read, Update and Delete actions.

The code to implement this functionality should be in its own class – for example, the Db class – so we can hide the details of the database from the rest of our application. The singleton class pattern is a good one to use here because there is only one database, and a singleton class will give our entire application access to it without having to create multiple instances. We could, in fact, create multiple instances but each instance would require its own database connection. This is not necessarily bad, but each connection would need to be opened and closed – an expensive operation. Also, each connection is independently processed by the database and that could cause synchronization problems for us in complex situations. In a desktop application, sharing a single connection eliminates these issues.

In web applications where multiple, independent, users are accessing the application simultaneously through a web server, sharing a single connection would be a performance bottleneck as users would need to wait their turn before being returned data. This tips the scales and makes the additional complexity of multiple connections warranted. Even here, though, the singleton class could manage a pool of connections that are opened once and distributed as needed to service the various user requests.

Adding a connection string

The first thing we need to do is to introduce a new file type in a C# project, the .config file. To add this file to your project, right click on the project in the Solution Explorer window, select Add... then New Item... In the dialog that is now shown, select 'General' from the left-hand list. Now select 'Application Configuration File' from the right-hand list. There is no need to change the default name of App.config. With that file open, change its contents to look like:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add connectionString="Server=.\SQLEXPRESS;Database=NorthWind;Trusted_Connection=True;"
        name="conn"/>
  </connectionStrings>
</configuration>
```

While we could embed the connection string directly into our app as a constant, adding it as a connectionString element to this file is better because if we need to run this application on another computer, we can update the .config file generated by the build process to the new value without having to rebuild the application for each computer. This is not something you would want to do for an application that will be distributed to millions of users, but it is a simple way of separating configuration information from the code in more limited scenarios where only a few copies of the application will ever be deployed.

The value of the connectionString attribute can, in many cases, be the most difficult part to get right. If you visit <https://www.connectionstrings.com/sql-server/> you will find a very long list of possible connection strings for SQL Server, as well as other databases, you may encounter in the field. Depending on where the hardware running your SQL Server is located compared to where your application is running, you may need to provide a network address and login credentials. In this case, though, we will assume that the computer running your application is the same one running the SQL Server instance your code will want to communicate with, as this is the most straightforward.

Looking at the string, the first element is the 'Server' component. If SQL Server was installed on your computer with the default options, the SQL Server instance we want to access is on our machine – as indicated by the '.' – and has the name SQLEXPRESS. It is possible, however, that the name is different. To find the correct name, start SSMS and check the initial login dialog. The text it populated into the Server Name field will typically be in the form of <computer name>/<sql server name>. You could just use that string directly, but the period replaces the computer name with the name of the computer the application is currently running on – making it possible to run this application on multiple machines without rebuilding.

The second part of the connection string identifies the name of the database we want to connect to. If you are following along, then we will be working with a database named NorthWind.

Finally, the 'Trusted Connection' setting, when set to true, indicates that we want to use Windows Authentication. That is, we are asking it to use the currently logged in user's credentials to connect to the database. In fact, that is the default for SSMS as well. This is generally fine for testing, but in production situations, we may want to replace this with a username/password if our application is not installed on the same computer that is running the database server.

Creating a database class

The basic structure of a DB singleton class could look like:

```
private static Db db;
public static Db Inst => db ??= new Db();

private readonly SqlConnection conn;
private Db() => (conn = new SqlConnection(
    ConfigurationManager.ConnectionStrings["conn"].ConnectionString)
)?.Open();
```

Here, we use the typical singleton pattern to create a single instance of the Db class and return it via the Inst static property. The ??= syntax in the Inst property allows us to write a shortened version of

```
public static Db Inst
{
    get
    {
        if (db == null)
            db = new Db();
        return db;
    }
}
```

Which ensures that a new Db instance is created only if one does not yet exist.

The last line combines a few separate actions into a single statement. First, this

```
ConfigurationManager.ConnectionStrings["conn"].ConnectionString
```

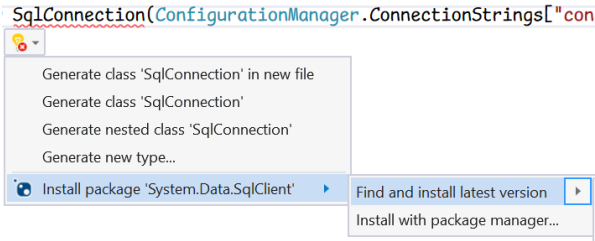
Reads the connection string tag we created earlier in App.config. Then it uses that string to create a new SqlConnection object and saves it to the private variable 'conn'.

```
conn = new SqlConnection(...)
```

Finally, it opens the connection if it is not null

```
(...)?.Open()
```

When you first write this code, however, Visual Studio will indicate that it does not know about the SqlConnection class. To add this library, we need to add a reference to the class and add a using statement. Depending on the version of .NET you are using, clicking on the help may give you this option



If so, select the 'Find and install latest version' option. It will add the reference and also add this using statement:

```
using System.Data.SqlClient;
```

It is possible to have more than one connection open to a database, but the number of connections is not infinite. What we are doing here is opening a single connection that will stay open for as long as the application is running. It is also possible to open and close connections as needed. This requires additional bookkeeping, so we will not do it here, but there may be circumstances (many people running your app at once) where releasing connections when not needed is appropriate. Instead, we will rely on the Garbage Collector to clean up the connection for us when the application closes.

Now that we have a connection, we need to use it.

Reading records

First, we need to create an object to represent the database table in our app. If you go to SSMS, create a new query, enter this, and execute it:

```
select * from customers
```

You will see a list of customer records. Each column represents a property in the C# version of this record, as follows:

```
internal class Customer
{
    public string CustomerID { get; set; }
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
    public string Fax { get; set; }
}
```

Note that, just like in the database where all columns are either `nchar` or `nvarchar` types, in the C# version all properties are strings. That includes phone numbers, which despite the name we call them are not actual numbers. A rule of thumb when designing databases is that if you never expect to invoke a mathematical function on a value, it should be stored as a string. Since it makes no sense to add, subtract, multiply, divide or calculate an average of a phone number, then it should be stored as a string.

Now that we have this mapping, we can write some code to create a list of customers from the database.

```
const string SELECT_CMD_TEXT = "SELECT * FROM Customers";

internal List<Customer> GetCustomers()
{
    List<Customer> customers = new List<Customer>();

    using var cmd = new SqlCommand(SELECT_CMD_TEXT, conn);
    SqlDataReader dr = cmd.ExecuteReader();

    while(dr.Read())
    {
        customers.Add(new Customer
        {
            Address = dr.GetString(dr.GetOrdinal("Address")),
            City = dr.GetString(dr.GetOrdinal("City")),
            ...
            Region = dr.GetString(dr.GetOrdinal("Region")),
        });
    }
    dr.Close();

    return customers;
}
```

To get the list of customers, we use the SQL statement from our test in SSMS. To have C# execute the SQL statement for us, we need to create and use an `SqlCommand` object. There are multiple variations of the constructor for this class, but the one used in the sample code is the one I most commonly use as it is expressed in one line of code. You could equivalently do:

```
using var cmd = new SqlCommand
{
    CommandText = SELECT_CMD_TEXT,
    Connection = conn
};
```

If you would rather follow the new style of object initialization. In either case, you will have noticed the use of the 'using' keyword in a new context. Normally we see this to specify which libraries the code is using. In this context, though, it has a different meaning. It is informing the runtime when to dispose of the object just created. Without it, the garbage collector will eventually clean up the `SqlCommand` object. However, there are a limited set of `SqlCommand` objects that can be created. So it is possible to run out in a busy application, which would cause intermittent failures until the garbage collector cleaned things up.

The command must be executed to retrieve results from the database. Because we are reading data, the `SqlCommand` method we use to do the reading is the `ExecuteReader()` method. This creates a `SqlDataReader` object that allows us to iterate over the results returned by the database using its `Read()` method. Note that you could also create a `DataSet()` object instead of a `SqlDataReader`. This is no longer a recommended approach as it is referred to by Microsoft as a 'legacy' API, making it something that could be removed in future versions.

We loop as long as `Read()` tells us there's another record (by returning `true`). Each time through the loop we create and add a new `Customer` object to our list. To map a column's data to the object's property, use the `SqlDataReader`'s `Getxxx()` methods. In this case, all the fields are strings, so we use the `GetString()` method exclusively. These methods read one field at a time and use an integer index to identify the field to be read. Because we used `'select *'`, we don't necessarily know the order of the fields that `Sql Server` will return. To solve that, we use the `SqlDataReader`'s `GetOrdinal()` method to find the index of the field whose name we pass in. It is also possible to write

```
const string SELECT_CMD_TEXT = "SELECT Address, City, Region FROM Customers";
```

If we call out the order of the fields explicitly, then we can write

```
Address = dr.GetString(0)
```

And use the index directly. There is not much difference between the two approaches for small numbers of records and small numbers of columns. However, if the number of columns in the table is large, then keeping track of the order of fields named in the select can introduce errors. If the number of records is large, then calling the `GetOrdinal()` method hundreds or thousands of times can have a noticeable effect on the application's performance. This can be solved by calling the `GetOrdinal()` method once for each column outside the while loop.

If there are no customers, we will return an empty list. Once we have this list, we can use the LINQ components to filter and sort our data. We could also use SQL itself to filter and sort data. Which to pick can be a personal preference for certain applications. However, there are some things to keep in mind:

- If the SQL server instance is remote from your application (like over the internet), then sending many queries will slow down the application, since it must wait for network delays before receiving the response. This would tend to favor having the app request all records and filtering them locally.
- If many users are simultaneously accessing the same SQL server instance, then having them all generate many queries will add even more delay to the system. Again, this would tend to favor local filtering via Linq.
- In a large system, the amount of data returned from retrieving all records from all tables may overwhelm the application and cause poor performance. This tends to favor filtering records in SQL and working with a subset only.
- In a system with many users capable of writing to the database, data cached locally may be out of date at any time. This would tend to favor having SQL do the filtering.

There is no one correct way to approach this problem and each application must be examined on its own merits. As well, there are ways to mitigate these issues. If, for instance, the number of records in the database is large, but only some of them apply to the current user, then doing that part of the filtering in SQL followed by local filtering is a reasonable approach.

Dealing with data caching issues is one of the unsolved problems of computer science. But one way to keep things approximately up to date is to associate a timestamp with each query and reacquire the data only if a certain amount of time has passed since the last query. How long that time should be depends on the application's sensitivity to being out of date. The exact number of 'likes' that an Instagram post has is not crucial information and can tolerate being somewhat out of date for

minutes. If the inventory count of a hot selling product is out of date by minutes, it could mean hundreds or thousands of disappointed customers who will go through most of the purchasing process only to find that their order will not be shipped.

Updating records

To update a record in SQL, you would write:

```
update customers set PostalCode = '42101' where CustomerID = 'REGGC'
```

Notice a couple of things with the line above. First, strings in sql are bracketed by the single quote character and, second, the single equal character indicates both comparison and assignment. Also note that it is extremely important to include the 'where' clause in update statements. Otherwise, all records in the table will have the specified column set to the same new value.

The C# version of this SQL statement looks like:

```
const string UPDATE_CMD_TEXT = "UPDATE Customer SET " +
    "address=@ADDRESS," +
    "city=@CITY," +
    "companyName=@COMPANYNAME," +
    "contactName=@CONTACTNAME " +
    "contactTitle=@CONTACTTITLE " +
    "country=@COUNTRY " +
    "fax=@FAX " +
    "phone=@PHONE " +
    "postalCode=@POSTALCODE " +
    "region=@REGION " +
    "WHERE customerID=@CUSTOMERID";

internal bool UpdateCustomer(Customer c)
{
    using var cmd = new SqlCommand(UPDATE_CMD_TEXT, conn);
    cmd.Parameters.AddWithValue("@ADDRESS", c.Address);
    cmd.Parameters.AddWithValue("@CITY", c.City);
    cmd.Parameters.AddWithValue("@COMPANYNAME", c.CompanyName);
    cmd.Parameters.AddWithValue("@CONTACTNAME", c.ContactName);
    cmd.Parameters.AddWithValue("@CONTACTTITLE", c.ContactTitle);
    cmd.Parameters.AddWithValue("@COUNTRY", c.Country);
    cmd.Parameters.AddWithValue("@CUSTOMERID", c.CustomerID);
    cmd.Parameters.AddWithValue("@FAX", c.Fax);
    cmd.Parameters.AddWithValue("@PHONE", c.Phone);
    cmd.Parameters.AddWithValue("@POSTALCODE", c.PostalCode);
    cmd.Parameters.AddWithValue("@REGION", c.Region);
    int count = cmd.ExecuteNonQuery();

    return count > 0;
}
```

When writing custom SQL by hand, we can tailor it to only specify the fields we know are changing. However, we are not keeping track of which fields in our C# object have been updated, so we cannot take the same approach in our C# code. Instead, we update all fields every time. Updating a field to the same value as it previously held costs us nothing, so there is no performance penalty to be paid – unless the field contains a large amount of data. In that case, creating a separate update method whenever the

large field changes might be warranted. Of course, we should not modify the CustomerID field as it is the primary key.

This time we call `ExecuteNonQuery()` to run the SQL statement because we do not expect the result to be a set of records. `ExecuteNonQuery` returns a count of the number of affected records. If that count is more than zero, we did an update. We could also check for exactly one record affected, but since IDs are unique in this database, we know that it would not be possible for more than one record to be changed by this statement. So, checking for the result to be greater than zero is sufficient.

To provide the data to the query, we use what is known as parameterized queries. We do not construct the SQL statement directly with string concatenation, but instead with parameters that we provide to the `SqlCommand` `Parameters` object. This takes the form of providing a variable in the command string - `@CITY` for instance – and then making an association with that variable by calling the `AddWithValue()` method like so

```
cmd.Parameters.AddWithValue("@CITY", c.City);
```

This associates the variable with the string from the object that holds the name of the city.

It is important to use this approach whenever user generated data is to be written to the database because .NET will sanitize the inputs provided via parameters and ensure that they are not interpreted as SQL statements. For example, this code:

```
string pc = "'7'; DROP TABLE Customers; --";
string sqlCmd = $"UPDATE Customers SET postalCode = {pc} WHERE customerID = 'REGGC'";
```

intends to update the postal code. However, the user provided string 'pc' does not contain a normal postal code, but instead a fragment of SQL that SQL Server will interpret as these two commands plus a comment:

```
UPDATE Customers SET postalCode = '7';
DROP TABLE Customers;
-- WHERE customerID = 'REGGC'
```

First, it sets all the postal codes in the table to the string '7'. Then it deletes the entire customer table. Finally, it takes any remaining SQL that might result in a syntax error and converts it to a comment since in SQL `--` is the start of a comment. Clearly not what was desired, but possible if the user knows the right things to type into a text box on our system.

With that same input but using parameters, the string `"'7'; DROP TABLE Customers; --"` is stored in the postal code field for the record with customerID 'REGGC' – which does exactly what we would expect our code to do.

Also note that with the string concatenation version of the code, we need to worry about putting single quote characters around all the strings. SQL syntax errors will result if we do not. However, the parameterized version populates the SQL query it generates appropriately for the data type. However, do note that in both cases it is our job to ensure the spaces between keywords are maintained.

Deleting records

Deleting a record in SQL looks like this:

```
const string DELETE_CMD_TEXT = "DELETE FROM Customers WHERE customerId=@CUSTOMERID";
```

```

internal bool DeleteCustomer(Customer c)
{
    SqlCommand cmd = new SqlCommand(DELETE_CMD_TEXT, conn);
    cmd.Parameters.AddWithValue("@CUSTOMERID", c.CustomerID);
    int count = cmd.ExecuteNonQuery();

    return count > 0;
}

```

Again we use parameterized statements and check the return value from `ExecuteNonQuery()` to ensure the record was found and deleted.

Depending on the application being developed, we may not actually want to ever delete records. Instead, we can mark them as deleted or inactive by adding a column to indicate the state of the record. The simplest approach is to use a Bit data type column. If you call the column 'Deleted', then zero indicates a valid record and one indicates that it has been deleted. A common approach is to use a DateTime field initially set to null. If the field is null, then the record is valid. When it is time to delete the record, then set the field to the time of deletion. This provides additional information that could come in handy when trying to debug a problem. You could also add the name of the user who deleted for a more complete audit trail. Additionally, adding similar DateTime fields for record creation and record update provides even more audit information.

For example, the NorthWind database has no way of indicating that a customer has gone out of business and should no longer be shown in a list of active customers. However, we cannot just delete the customer's record from the database. Any records in the Orders table associated with that customer in the past would lose their associated customer record and would be orphaned. A report intended to show historical order information would then show that an 'unknown' customer placed some of the orders. We should not clean that up by deleting orders from the database. If we did, we could be accused of tax fraud for reducing our revenue.

Creating records

The SQL for creating a new record in a table looks like this:

```

insert into Customers (CustomerID, CompanyName, ContactName, ContactTitle, Address,
City, Region, PostalCode, Country, Phone, Fax)
values ('AAAAA', 'ACME Razors', 'John Smith', 'Owner', '1 Main St.',
'Anytown', 'OH', '00000', 'USA', '555-555-1212', '555-555-1213')

```

It specifies the table, then a bracketed list of the column names in order, and then after the 'values' keyword, a bracketed list of the values (in this case, all strings)

And the C# version looks like:

```

const string VALIDATE_CMD_TEXT = "SELECT COUNT(*) FROM Customers " +
                                "WHERE CustomerID = @CUSTOMERID";
const string INSERT_CMD_TEXT = "INSERT INTO Customers " +
    "(address, city, companyName, contactName, contactTitle, " +
    "country, customerId, fax, phone, postalCode, region) " +
    "VALUES " +
    "(@ADDRESS, @CITY, @COMPANYNAME, @CONTACTNAME, @CONTACTTITLE, " +
    "@COUNTRY, @CUSTOMERID, @FAX, @PHONE, @POSTALCODE, @REGION)";

```



```

internal bool AddCustomer(Customer c)
{
    var success = false;

    try
    {
        var cmdV = new SqlCommand(VALIDATE_CMD_TEXT, conn);
        cmdV.Parameters.AddWithValue("@CUSTOMERID", c.CustomerID);
        int existingRows = (int)cmdV.ExecuteScalar();

        if (existingRows == 0)
        {
            var cmdI = new SqlCommand(INSERT_CMD_TEXT, conn);
            cmdI.Parameters.AddWithValue("@ADDRESS", c.Address);
            cmdI.Parameters.AddWithValue("@CITY", c.City);
            cmdI.Parameters.AddWithValue("@COMPANYNAME", c.CompanyName);
            cmdI.Parameters.AddWithValue("@CONTACTNAME", c.ContactName);
            cmdI.Parameters.AddWithValue("@CONTACTTITLE", c.ContactTitle);
            cmdI.Parameters.AddWithValue("@COUNTRY", c.Country);
            cmdI.Parameters.AddWithValue("@CUSTOMERID", c.CustomerID);
            cmdI.Parameters.AddWithValue("@FAX", c.Fax);
            cmdI.Parameters.AddWithValue("@PHONE", c.Phone);
            cmdI.Parameters.AddWithValue("@POSTALCODE", c.PostalCode);
            cmdI.Parameters.AddWithValue("@REGION", c.Region);
            int count = cmdI.ExecuteNonQuery();

            success = count > 0;
        }
    }
    catch { }

    return success;
}

```

For this database table, adding a record is a two-step process. This table has a primary key – a value that must be unique – that is a string the user can set. Because of this, we must check the key to ensure it is not already in the database. That is what `VALIDATE_CMD_TEXT` does. It counts the number of records with that key. Although we could have used `ExecuteReader` to run this SQL statement against the database, we use `ExecuteScalar()` instead because, in this case, we are expecting only a single value to be returned. `ExecuteScalar` provides a more convenient mechanism to access that single returned value from the `SELECT` query.

If none were found that match, we can go ahead and insert this record. Again, we use parameters and `ExecuteNonQuery()`. We also wrap this in a `try/catch` in case something goes wrong. If an exception does get thrown, it will return `false` and the caller will know that the record was not written to the database.

Transactions

Most of the time, updating the database is done one table at a time without dependency on other tables. However, this is not always the case. In the `NorthWind` database, for example, the `OrderDetails` table has a foreign key dependency on the `Orders` table. It is easy to image a UI that lets us create an `Order` object and then one or more `OrderDetails` objects. When the user presses the ‘Save’ button in the

UI, all these objects need to be written to the database. Since we do not want to save a partial set of objects, we could wrap all the writes in a transaction that is rolled back if the writing of any object fails. If that occurs, we can go back to the user and let them know that the order failed to save entirely – rather than saving a fraction of the order and then forcing the user to add the missing pieces manually.

For example, if our app required that we be able to delete orders – those not yet processed, for instance – then we might write code that looked like:

```
const string DELETE_ORDER_TEXT = "DELETE FROM Orders WHERE orderId = @ORDERID";
const string DELETE_ORDERITEMS_TEXT = "DELETE FROM [Order Details] WHERE orderId = @ORDERID";

internal bool DeleteOrder(Order o)
{
    var count = 0;
    using var t = conn.BeginTransaction();

    try
    {
        using var cmdI = new SqlCommand(DELETE_ORDERITEMS_TEXT, conn, t);
        cmdI.Parameters.AddWithValue("@ORDERID", o.Id);
        cmdI.ExecuteNonQuery();

        using var cmdO = new SqlCommand(DELETE_ORDER_TEXT, conn, t);
        cmdO.Parameters.AddWithValue("@ORDERID", o.Id);
        count = cmdO.ExecuteNonQuery();

        t.Commit();
    }
    catch
    {
        t.Rollback();
    }

    return count > 0;
}
```

We create a new `SqlTransaction` instance and pass it as a third parameter when creating the `SqlCommand` objects. Since the `ExecuteNonQuery` will throw an exception if something goes wrong, we can put all of this inside a `try/catch`. If an exception is thrown, then we can roll back the transaction with the `Rollback()` method. If all went well, then calling `Commit()` commits the changes to the database. We could also have written a single SQL statement that handled all of this in one call to `ExecuteNonQuery`, but the advantage of a transaction object in C# is that we can pass it around as a parameter to multiple methods, some of which can be called optionally. Depending on your comfort level with writing SQL, it may be easier to use C# to describe the logic of complex chains of SQL statements.

Exercise 13

Use all the knowledge you have gained so far to create an application that uses the NorthWind database to display, create, delete, sort and filter orders and order details. You will need to create classes for each of the relevant elements: customers, orders, order details, products, employees, and shippers. There is, however, no need to create a full product object for this application. Only `ProductId`, `ProductName`, `QuantityPerUnit`, and `UnitPrice` fields are necessary – assume an infinite supply. For employees, only their id and name fields are necessary. When completed, compare your version with what is found in Exercise 13.

Asynchronous Execution

Each application running on your computer is running as a separate process. One of the jobs of an operating system like Windows is to mediate between processes so they share the CPU and other resources – known as multitasking. Thus, your music player and your word processor can both be doing work at the same time. In early versions of Windows, the multitasking was co-operative. That is, Windows depended on applications to be good citizens and give up the CPU on occasion to allow other processes to run. This generally worked, but if an application accidentally implemented an infinite loop, then all applications would come to a stop. The current version of Windows uses something called preemptive multitasking. In this case, Windows lets each application run for a limited amount of time and then forces it to wait until other applications have a chance to run. Now, an infinite loop in one application only affects that application and no other. The job of scheduling applications is more complex than ever with multi-core CPUs, but the concepts are the same.

Under the covers on startup of a process, it creates a thread object. This thread object is what Windows is really scheduling. A process can create more than one thread and it can give threads varying priorities that will determine how often Windows provides the thread with CPU time.

In all the applications we have created so far, there has only been one thread – usually referred to as the main or UI thread. Since we only have the UI thread to work with, all the work we do is run on that thread. Any time we write some code that takes a noticeable amount of time to run, the user will notice the application stop responding until our code finishes its work. That is, a click on a button with event handling code that has an infinite loop means that no other button can ever be clicked in our application.

Because we do not want the user to experience delays, we should create new threads for any long running work. This is especially important if that long running work might need to be aborted. If the UI thread becomes unresponsive, then the user will not be able to stop the operation and now must wait for it to finish and then discard the results.

Once the long running task is on that new thread, Windows will allocate it CPU time according to its priority along with the UI thread – meaning that the UI thread will still be responsive.

BackgroundWorker

One way that .NET offers to get another thread to do some long running task is the BackgroundWorker class. To set up a BackgroundWorker, you could write code in your VM class like:

```
worker = new BackgroundWorker
{
    WorkerReportsProgress = true,
    WorkerSupportsCancellation = true
};
worker.DoWork += worker_DoWork;
worker.ProgressChanged += worker_ProgressChanged;
worker.RunWorkerCompleted += worker_RunWorkerCompleted;
worker.RunWorkerAsync(50);
```

In this case, the BackgroundWorker object is initialized to allow us to report progress and to cancel the operation at any time. To start the whole process, we call the RunWorkerAsync() method. With this method you can optionally provide an object parameter that DoWork can access.

As we have seen before, the += implies that the BackgroundWorker object we have created has events that we can register to receive. The DoWork event lets us know that we can start doing the long running work. The ProgressChanged event is triggered by us from within DoWork whenever we want to update some progress indicator in the UI. The RunWorkerCompleted event is triggered when the DoWork event handler method returns. In this simple example, we count with some delay and update the UI elements via the MVVM pattern:

```
private int progressPct;
public int ProgressPct { get => progressPct; set => SetField(ref progressPct, value); }

public BindingList<string> Values { get; set; } = new BindingList<string>();

private bool isComplete;
public bool IsComplete { get => isComplete; set => SetField(ref isComplete, value); }

private void worker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    IsComplete = true;
}

private void worker_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    ProgressPct = e.ProgressPercentage;
    Values.Add(e.UserState?.ToString());
}

private void worker_DoWork(object sender, DoWorkEventArgs e)
{
    var max = e.Argument as int? ?? 0;
    var scale = 100.0 / max;
    var prevProgress = 0;

    for (var i = 1; i <= max; i++)
    {
        if (worker.CancellationPending)
            break;
        Thread.Sleep(500);

        var curProgress = (int)(scale * i);
        if (curProgress != prevProgress)
        {
            worker.ReportProgress(curProgress, i);
            prevProgress = curProgress;
        }
    }
}
```

In this case, worker_RunWorkerCompleted sets the IsCompleted boolean which causes the XAML to display something to the user. The worker_ProgressChanged method updates both the ProgressPct property and the Values BindingList. Finally, worker_DoWork does the actual calculations – in this case simply counting to the value provided in the parameter to RunWorkerAsync() and accessed via e.Argument but with Thread.Sleep(500) delaying each count by half a second. At the end of each loop, we check to see if the progress has changed and call the ReportProgress() method. This method takes

two parameters, first the progress percentage and second some object that, in this case, we add to the `BindingList` in `ProgressChanged`.

Note that calling `ReportProgress` too quickly will result in the UI thread appearing to stop working, so do not overwhelm it with updates.

Also note that we do not – and cannot – update the UI directly from within `DoWork`. If you do, an exception will be thrown like:

System.NotSupportedException: 'This type of `CollectionView` does not support changes to its `SourceCollection` from a thread different from the `Dispatcher` thread.'

The reason for this is that other threads cannot access UI elements that live on the UI thread. `ProgressChanged` and `RunWorkerCompleted` can change UI elements because they run on the UI thread. The `BackgroundWorker` instance is doing the communications from one thread to the other on our behalf.

Finally, to cancel the thread early, we are checking the `CancellationPending` boolean each time through the loop. This gets set by writing this code:

```
worker.CancelAsync();
```

and executing it on the UI thread. It does not immediately stop the thread, but only sets the boolean. It is up to the `DoWork` code to check `CancellationPending` often enough to prevent long delays before the cancelation takes effect.

Thread

The `BackgroundWorker` is fine if you have some operation that runs and then completes, but if your application requires a background task to run for the duration of your application, then it is better to take a different approach. For example, we might have a background thread that sends emails from our application. Making the user wait for the whole email process to complete could take a long time – especially if the computer is not connected to the network.

A better approach is to send a request to the thread to add a new email to the queue of emails to be sent. If everything is normal, then the one email will be sent immediately. But if things are not normal, then the request can be stored in a database table for later and sent when a connection is reestablished. In any case, the user is not delayed and can move on to their next task.

We can also recreate `BackgroundWorker` using the `Thread` class using a class that looks something like:

```
internal class ThreadObject
{
    private Thread worker;

    public Action<object> DoWork { get; set; }

    public bool IsCancelRequested { get; private set; }

    public void Run(object o)
    {
        if (worker == null || worker.ThreadState != ThreadState.Running)
        {
```

```

        //create a new background worker
        worker = new Thread(worker_DoWork);
        worker.Start(o);
    }

    public void Cancel()
    {
        IsCancelRequested = true;
    }

    private void worker_DoWork(object o)
    {
        DoWork(o);
    }
}

```

In this code, the Run() method is equivalent to BackgroundWorker's RunWorkerAsync() method. The Cancel() method maps to BackgroundWorker.CancelAsync(). Rather than using event handlers for progress, completion, and calculations, this class exposes a DoWork property that the user of the class can populate with a method to do the calculations. That results in a somewhat modified version of the code that uses this class. First, we initialize and begin background execution like this:

```

private ThreadObject thread;
public void Start()
{
    if (thread == null)
    {
        //get the UI back in starting mode
        IsComplete = false;
        ProgressPct = 0;

        //create a new background worker
        thread = new ThreadObject
        {
            DoWork = doWork
        };
        thread.Run(50);
    }
}

```

That is, with a similar approach to that of background worker. The doWork() method now looks like:

```

private void doWork(object o)
{
    var max = o as int? ?? 0;
    var scale = 100.0 / max;
    var prevProgress = 0;

    for (var i = 1; i <= max; i++)
    {
        if (thread.IsCancelRequested)
            break;
        Thread.Sleep(500);

        prevProgress = onProgress(scale, prevProgress, i);
    }
}

```

```

        onCompletion();
    }

```

Which is again similar, except that rather than making calls to public methods that then post events, we call the local methods directly. This is where the main difference lies, because our local methods are now running on the background thread and cannot directly manipulate the UI elements. So our local methods for completion and progress must look like this:

```

private readonly Dispatcher dispatcher = Dispatcher.CurrentDispatcher;

private void onCompletion()
{
    dispatcher.BeginInvoke(new Action(() =>
    {
        //reset things
        IsComplete = true;
        thread = null;
    }));
}

private int onProgress(double scale, int prevProgress, int value)
{
    var curProgress = (int)(scale * value);
    if (curProgress != prevProgress)
    {
        dispatcher.BeginInvoke(new Action(() =>
        {
            ProgressPct = curProgress;
            Values.Add(value.ToString());
        }, null));
        prevProgress = curProgress;
    }

    return prevProgress;
}

```

The main difference is our explicit use of the `Dispatcher.CurrentDispatcher` object to transfer control to the UI thread for code that potentially manipulates UI elements. This is what `BackgroundWorker` is doing for us internally so that our event handlers run on the UI thread.

Task

A third approach to creating additional threads is the `Task` class. This is a higher level concept that uses the `Thread` class internally. The main difference between using `Task` and `Thread` is that `Task` uses threads from the thread pool that .NET provides each process whereas `Thread` always creates a new thread. Since creating and destroying threads is expensive, using the `Task` class is preferred unless there is some specific feature of `Thread` that is not available in `Task`. For instance, in the `Thread` code above, the `IsBackground` property defaults to `false`, and so the thread we create is a foreground thread. A thread from the thread pool, on the other hand, is always a background thread. This provides some advantages as Windows will provide more CPU time to a foreground thread. When using `Task`, we do not need to create a `ThreadObject` class as we did above. Instead, we can write:

```

private Task bkgTask;
private CancellationTokenSource cancellationTokenSource;

```

```

public void Start()
{
    if (bkgTask == null)
    {
        //get the UI back in starting mode
        IsComplete = false;
        ProgressPct = 0;

        //create a new background worker
        cancelTokenSource = new CancellationTokensource();
        bkgTask = Task.Run(() => doWork(50, cancelTokenSource.Token));
    }
}

```

to start the task using `Task.Run()`. The code is much the same except that now we can pass a method to `Task.Run` that runs our `doWork()` method directly. Since we are running it directly, we can specify the exact data type in the parameter list and customize the parameters as needed without having to go through a generic interface. The second parameter is the token property of the `CancellationTokensource` class. In this case, we will use that .NET class to implement the cancel feature.

To cancel, we can now write this:

```

public void CancelCalc() => cancelTokenSource.Cancel();

```

and we check for this in the `doWork()` method by

```

private void doWork(int max, CancellationTokens token)
...
    if (token.IsCancellationRequested)
        break;

```

but otherwise, the implementation is the same, including using the dispatcher object to update UI elements. Given the simplicity of this approach and the advantages of using the thread pool, most asynchronous operations should be implemented this way.

Exercise 14

Create an application that will find the largest prime number that is less than a number the user enters. Allow the user to cancel the search before it is completed. Also handle a button click and display a message to demonstrate that the UI thread responds while calculations are done. Use one of `BackgroundWorker`, `Thread`, or `Task`. When completed, compare your version with what is found in Exercise 14.

Async/Await

Up to this point in the discussion, we have focused on examples where the background thread is doing some CPU intensive task. It is more common, however, for the CPU to be idle and waiting for some other device to respond to a request. For example, delays can be long, from the CPU's perspective, when reading or writing to a file. A typical SSD SATA drive can read data at approximately 500 MB/s. That data must be copied into RAM before the processor (and your code) can access and manipulate it. If you had a 10MB file that you wanted to process, that would take 20msec to get the file into memory. Typically, this transfer is done using Direct Memory Access (DMA) hardware with little CPU involvement. During that time the CPU could be doing other work. However, if the call is made synchronously like this:


```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string text1 = File.ReadAllText("filePath");
    string[] lines = text1.Split('\n');
}
```

then the UI thread is blocked for the entire duration of the read. However, if we were to write our own asynchronous ReadAllText, like this:

```
public async Task<string> ReadAllTextAsync(string filePath)
{
    using var fileStream = File.OpenRead(filePath);
    using var streamReader = new StreamReader(fileStream);
    var stringBuilder = new StringBuilder();

    string line;
    for (;;)
    {
        if ((line = await streamReader.ReadLineAsync()) == null)
            break;
        stringBuilder.AppendLine(line);
    }

    return stringBuilder.ToString();
}
```

And call it this way

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    string text2 = await ReadAllTextAsync("filePath");
    string[] lines = text2.Split('\n');
}
```

Then we can avoid those delays. Note, of course, that the File class already has a ReadAllTextAsync, so writing this code is not necessary.

However, in our version of the code, you will have noticed two new keywords: `async` and `await`. The `async` keyword is used in the method declaration line to indicate that the code will contain asynchronous method calls. The `await` keyword tells the computer *to go do something else while this asynchronous method is working and, when it is done, come back here and continue to execute this code*.

At the lowest level of this code, `streamReader.ReadLineAsync()` is an asynchronous method that will take some time to read the next line from the file. We want to do something with the line of text that comes back so we cannot do any other work until we get it. But we don't want the thread to block while we wait. The `await` keyword causes all the heavy lifting of ceding control back to windows to be handled for us. At the next level, we also want to wait for the results of `ReadAllTextAsync()` before we parse the text into lines, so again, we use the `await` keyword to make this happen. We could also use this form:

```
string text2 = ReadAllTextAsync("filePath").Result;
```

because the return data type of `ReadAllTextAsync()` is a `Task<string>` and a task object has a `Result` property. The difference in the two approaches is that using the `Result` property does not require the `await` keyword, and so the method does not require the `async` keyword in its signature. As you may have

noticed, adding an await can bubble all the way back up the call chain unless you use something like Result or Wait. Note, however, that using Result and Wait cause the thread to block, so the advantages using asynchronous methods are lost. In general, this should be avoided, but there are circumstances, like inside the constructor for a class, for instance, that do not allow asynchronous methods, so Result and Wait are required. Code inside the set or get of a property also does not support async methods. However, in this case, Result and Wait will not work either.

One other point to mention is that typically the combination of async and void is not recommended in a method signature. If you do this, it is not possible for the caller to await the results. However, the one exception to this is in event handlers since they must return void. Otherwise, if your method does not return any data, have its signature be 'async Task'.

Another common situation where significant delays can occur is when accessing a database. In addition to the database's use of files, it is also possible that the database is running on a different computer and so network delays also need to be considered. If we take another look at the delete code from above:

```
internal bool DeleteCustomer(Customer c)
{
    SqlCommand cmd = new SqlCommand(DELETE_CMD_TEXT, conn);
    cmd.Parameters.AddWithValue("@CUSTOMERID", c.CustomerID);
    int count = cmd.ExecuteNonQuery();

    return count > 0;
}
```

Until the ExecuteNonQuery() gets a response from that networked database, we will sit waiting for that line of code to complete before moving on. During that time, the UI cannot respond to user actions. If the delay is long enough, the user could click multiple times on a button or scroll bar and then when the ExecuteNonQuery() is complete, all those queued actions are processed, resulting in unexpected behavior and frustrated users.

If, however, we use async/await syntax with the asynchronous version of the ExecuteNonQuery method, like this:

```
internal async Task<bool> DeleteCustomerAsync(Customer c)
{
    SqlCommand cmd = new SqlCommand(DELETE_CMD_TEXT, conn);
    cmd.Parameters.AddWithValue("@CUSTOMERID", c.CustomerID);
    int count = await cmd.ExecuteNonQueryAsync();

    return count > 0;
}
```

Then, instead of the normal call to this method, you would invoke it in this way:

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    bool isDeleted = await DB.Inst.DeleteCustomerAsync(customer);
}
```

You can think of this version as starting a new thread to wait for the `ExecuteNonQuery()` to complete. Because the waiting is being done on a new thread, the UI thread is free to continue processing user input, making the system more responsive.

To modify the original `DeleteCustomer` method, the signature was changed to add the `async` keyword. In addition, an `async` function does not return a data type, it now returns a `Task` object that promises to return the data type at some future point. Finally, we switch out the synchronous version of `ExecuteNonQuery` with its asynchronous version. The .NET framework has many methods with `sync` and `async` versions – basically for any time a call might be delayed because of some long duration I/O.

The `Task` object also has some additional methods to make our use of `async` and `await` more efficient. These are `WhenAll` and `WhenAny`. If we have a circumstance where we want to read from multiple files, we could write:

```
var text1 = await ReadAllTextAsync("filePath1");
var text2 = await ReadAllTextAsync("filePath2");
var text3 = await ReadAllTextAsync("filePath3");
```

and this will work. However, it will wait for each one to finish before starting the next read. If we wanted to take advantage of the parallelism provided by the File IO API, then we could instead write:

```
var text1task = ReadAllTextAsync("filePath1");
var text2task = ReadAllTextAsync("filePath2");
var text3task = ReadAllTextAsync("filePath3");
await Task.WhenAll(text1task, text2task, text3task);
string text1 = text1task.Result;
string text2 = text2task.Result;
string text3 = text3task.Result;
```

in this case, the three tasks are queued up and `WhenAll` returns control back once all three actions are done. If we only wanted to wait until one of them completed, then we could use `WhenAny` and `Task.IsCompletedSuccessfully` to figure out the result to take. It is also possible to collect up the tasks into an array and pass that to `WhenAll` or `WhenAny`.

Note that we have called `ReadAllTextAsync` with and without the `await` keyword. With the `await` keyword, the data type of the return value is a `string`. Without the `await` keyword used, it returns a `Task<string>` data type. It is very easy to forget the `await` keyword since neither will necessarily throw an error. However, they will act very differently because without the `await`, the method will not wait for the task to complete but will continue to immediately execute the next lines of code – which can cause various unexpected behavior.

Exercise 15

Modify the application you created in Exercise 13 to use asynchronous operations where possible. When completed, compare your version with what is found in Exercise 15.