Windows Phone 8 Programming in C#

Rob Miles
Windows Phone 8

Contents

Welcome to the wonderful world of Windows Phone development. If you have half as much fun reading this book as I've had writing it, then I've had twice as much fun as you. Which doesn't seem fair really.

Anyhow, I hope you find the content useful and enjoyable.

Rob Miles, September 2013
www.robmiles.com

1 Getting Started with Windows Phone

In this chapter you are going to find out about the Windows Phone platform as a device for running programs. You will learn the key features of the platform itself, how programs are written and also how you can sell your programs via the Windows Marketplace.

Programmer's Point: Stuff worth knowing

Every now and then I'm going to put these "Programmers Points" into the text. The aim of these is to give a professional developers perspective on the topic in hand.

1.1 The Windows Phone Platform

In this section we are going to take a look at the actual hardware that makes up a Windows Phone. This is particularly important as we need to put the abilities of the phone into context and identify the effect of the physical limitations imposed by platform that it uses.

A Windows Phone as a Computer

Pretty much everything these days is a computer. Mobile phones are no exception. When you get to the level of the Windows Phone device it is reasonable to think of it as a computer that can make telephone calls rather than a phone that can run programs.

The Windows Phone device has many properties in common with a "proper" computer. It has a powerful processor, local storage, fast 3D graphics and plenty of memory. It also has its own operating system which looks after the device and controls the programs that run on it. If you have used a PC you are used to the Windows operating system which starts running when you turn the computer and even turns the computer off for you when you have finished.

If you are familiar with computer specifications, then the specifications below are pretty impressive for a portable device. If you are not familiar, then just bear in mind that nobody in the world had a computer like this a few years ago, and now you can carry one around in your pocket.

The Windows Phone Hardware

Before we start programming we can take a look at the hardware we will be working with. This is not a text about computer hardware, but it is worth putting some of the phone hardware into context. All Windows Phones must have a particular minimum specification, so these are the very least you can expect to find on a device.

It is quite likely that different phone manufacturers will add their particular "take" on the platform, so you will find devices with more memory, faster processors, hardware keyboards and larger screens.

The Windows Phone Processor

The Central Processing Unit (CPU) of a computer is the place where all the work gets done. Whenever a program is running the CPU is in charge of fetching data from memory, changing the data and then putting it back (which is really all computers do). The most popular speed measure in a computer is the *clock speed*. A CPU has a clock that ticks when it is running. At each clock tick the processor will do one part of an operation, perhaps fetch an instruction from memory, perform a calculation and so forth.

The faster the clock speed the faster the computer. Modern desktop computers have clocks that tick at around 3 GHz (that is around 3 thousand million times a second). This is actually incredibly fast. It means that a single clock tick will last a nanosecond. A nanosecond is the time that light takes to travel around 30 cm. If you were wondering why we don't have

big computers any more, it is because the time it takes signals to travel around a circuit is a serious limiting factor in performance. Making a computer smaller actually makes it go faster.

A Windows Phone has a clock that ticks at a speed of at least 1GHz. You might think that this means a Windows Phone will run around a third the speed of a PC, but this turns out not to be the case. This is because of a number of things:

Firstly, clock speed is not directly comparable between processors. The processor in the Windows PC might take five clock ticks to do something that the Windows Phone processor needs ten ticks to perform. The Windows PC processor might be able to do things in hardware (for example floating point arithmetic) which the Windows Phone processor might need to call a software subroutine to perform, which will be much slower. You can regard clock speed as a bit like engine size in cars. A car with a bigger engine might go faster than one with a smaller one, but lots of other factors (weight of car, gearbox and tyres) are important too.

Secondly, a Windows PC may well have many processors. This doesn't mean a Windows PC can go faster, any more than two motorcycles can go faster than one, but it does mean they can process more data in a given time (two motorcycles can carry twice as many people as one). Windows 8 phones are available with multiple processors, but desktop PCs will usually have more.

Finally, a Windows PC has unlimited mains power. It can run the CPU at full speed all the time if it needs to. The only real problem with doing this is that the processor must be kept cool so that it doesn't melt. The faster a processor runs the more power it consumes. If the phone ran the processor at full speed all the time the battery life would be very short. The phone operating system will speed up and slow down the processor depending on what it needs to do at any given instant. Although the phone has a fast processor this speed is only actually used when the phone has something to do which requires very fast response.

The result of these considerations is that when you are writing a Windows Phone program you cannot regard processing power as an unlimited resource. Desktop PC programmers do not see processor speed as much of an issue but Windows Phone programmers have to remember that poor design can have consequences, both in terms of the experience of the user and the battery life of the phone. The good news for us is that worrying about these things will cause us to turn into better programmers.

Programmer's Point: Only Sweat the Speed when you have to

While it is true that a Windows Phone device has finite speed, and users always demand that your program provides results instantly, you should remember that actually the phone cpu is powerful. Most of the time your programs will probably run fast enough to provide a good user experience and when they don't it might be down to factors outside your control, such as network speed. This means that when I write a program I don't worry too much about the speed it runs at (although I'll avoid doing obviously stupid things that would make it run slowly). If it turns out that the a program is running too slowly I'll use profiling tools (which we will see later) to find out which bits are slowing the program down and take steps to speed them up .

The Windows Phone Operating System

Windows Phone 8 devices are based on the same operating system core as Windows 8 desktop PCs. Earlier version of Windows Phone (up to Windows Phone 7.51) were based on an embedded operating system called Windows CE (CE stands for "Compact Edition").

Some of the more advanced features, for example speech input, are only available for Windows Phone 8. However, you can write programs that can be run on both Windows 7 and Windows 8 devices.

Graphical Display

The Windows Phone has a high resolution display made up of a very large number of pixels. This provides good quality graphics and also allows lots of text to be displayed on the screen. The more pixels you have on your screen the higher the quality of image that you can display. However, the more pixels you have the greater the amount of memory that you need to store an image, and the more work the computer has to do to change the picture on the screen. This is particularly significant on a mobile device, where more work for the hardware translates to greater power consumption and lower battery life. The display resolution is a compromise between battery life, cost to manufacture and brightness of the display (the smaller the pixels the less light each can give out).

Windows Phone 8 devices are available in a number of different screen resolutions. The screen can be used in both landscape and portrait modes. The phone contains an accelerometer that detects how the phone is being held. The Windows

Phone operating system can then adjust the display to match the orientation. Our programs can decide when they run what orientations they can support. If we design our programs to work in both landscape and portrait mode they can be sent messages to allow them to adjust their display when the user changes the orientation of the device.

One problem faced by phone developers is the multitude of different screen sizes that are available. The Windows Phone emulator, which runs on your PC, allows you to preview applications running in different screen sizes. An application can also determine the screen dimensions and use these to scale the display. We will consider how to make "resolution independent" applications later in the text.

The Windows Phone Graphical Processor Unit

In the very first computers all the work was performed by the computer processor itself. This work included putting images on the display. Hardware engineers soon discovered that they could get faster moving images by creating custom devices to drive the screen. A Graphical Processor Unit (GPU) is given commands by the main processor and takes away all the work involved in drawing the screen. More advanced graphical processors have 3D support and are able to do the floating point and matrix arithmetic needed for three dimensions. They also contain *pixel shaders* which can be programmed to perform image processing on each dot of the screen at high speed as it is drawn, adding things such as lighting effects and blur.

Touch input

All Windows Phone devices have touch input which is capable of tracking at least four input points. This means that if you create a Windows Phone piano program it will be able to detect at least four notes being pressed at the same time.

The move to multi-touch input is an important step in the evolution of mobile devices. The user can control software by using multi-touch *gestures* such as "pinch". The Windows Phone operating system provides built in support for gesture recognition. Your programs can receive events when the user performs particular types of gesture.

Location Sensors

The Windows Phone device is location aware. It contains a Global Positioning System (GPS) device that receives satellite signals to determine the position of the phone to within a few feet. Since the GPS system only works properly when the phone has a clear view of the sky the phone will also use other techniques to determine position, including the location of the nearest cell phone tower and/or the location of the WIFI connection in use. This is called "assisted" GPS.

The Windows Phone operating system exposes methods our programs can call to determine the physical position of the device, along with the level of confidence and resolution of the result supplied. Our programs can also make use of mapping and search facilities which can be driven by location information. We will be exploring this later. The development environment also provides a really useful GPS emulator, so that you can test how your programs use position information without ever leaving your desk.

Accelerometer

The accelerometer is a hardware device that measures acceleration, so no surprises there. You could use it to compare the performance of sports cars if you wish. You can also use the accelerometer to detect when the phone is being shaken or waved around but the thing I like best is that you can also use it to detect how the phone is being held. This is because the accelerometer detects the acceleration due to gravity on the earth. This gives a phone a constant acceleration value which always points downwards. Programs can get the orientation of the phone in three axes, X, Y and Z.

This means we can write programs that respond when the user tips the phone in a particular direction, providing a very useful control mechanism for games. It is very easy for programs to obtain accelerometer readings, as we shall see later.

Compass

The phone also contains an electronic compass so that a program can determine which way the phone is facing. This can be useful in applications that are helping the user find their way around and in "augmented reality" where the phone overlays a computer drawn information on an image of the surroundings.

Gyroscope

A mechanical gyroscope is a device that always points in a particular direction. The Windows Phone contains an electronic version which allows it to detect when the phone is twisted and rotated. Programs can use the accelerometer to get an idea of how the phone is being held but a gyroscope gives much more precise information and can also give information about the rate of turn.

Sensor Integration and Simulation

It is great to have lots of sensors in the phone but there is a downside which is that the poor programmers (that's you and me) have create software that makes good use of all the different signals they produce. It can also be difficult to test how programs respond to a particular set of movements and actions. Fortunately the Windows Phone provides a single software object which integrates the information from the various sensors and provides a simple way a program can find out about the orientation and movement of the phone. This will make a best effort, depending on the sensors in the actual device. It is also possible to create particular actions and then have these replayed by the Windows Phone emulator when you are testing a program.

Camera

All mobile devices have cameras these days, and the Windows Phone is no exception. A phone camera will have at least 5 mega pixels or more. This means that it will take pictures that contain 5 million dots. This equates to a reasonable resolution digital camera (or the best anyone could get around five years ago). A 5 megapixel picture can be printed on a 7"x5" print with extremely good quality.

A Windows Phone application can take control of the camera and use it to take individual pictures or steams of video. It can also have access to the video stream itself, so that it can add things to the display to produce "augmented reality" applications or detect specific items such as barcodes or faces in photographs.

Developers can also create "Lens" applications which are integrated into the picture taking process and allow the user to apply effects and share pictures directly from the camera application in the phone.

Hardware buttons

All Windows Phone systems share a common user interface. As part of this design there are a number of physical buttons which are fitted to every Windows Phone that will always perform the same function, irrespective of the make or model of the phone.

Start: The start button is pressed to start a new activity. Pressing the start button will always take you to the program start screen, where you can select a new program and begin to run it. When you press the Start button this causes the currently running application to be paused (we will discuss this more a bit later). However, the Windows Phone operating system "remembers" which application was stopped so that you can return to it later by pressing the Back button.

Back: The back button is pressed to move back to the previous menu in a particular application. It is also used to stop one application and return to the one that was previously running. The back button makes the phone very easy to work with. You can start a new application (for example send an email message in the middle of browsing the web) and then once the message has been sent you press Back to return to the browser. Within the mail application the Back button will move you in and out of the various input screens and menus used to create and send the message. Once the message has been sent you can press Back at the top menu of the email application and return to the start menu and either run another program or press Back to return to browsing.

The Back button is also used to activate task switching on the phone. If you press and hold it down (the long press) the phone will present a display of all the active applications which you can move between.

Lock: Pressing the lock button will always lock the phone and turn off the display to save the phone battery. When the user presses the lock or start button again the phone will display the lock screen. A phone can be configured to automatically lock after a number of minutes of inactivity. It is possible to for an application to ask to be allowed to continue running "under" the lock screen. This is useful for applications such as navigation, where the program needs to remain active, but may have implications for battery life.

Search: Pressing the search button will start a new search. Precisely what happens when search is pressed depends on what the user is doing at the time. If the user presses search during a browsing session they will see a web search menu. If they press Search when the "People" application is active they can search for contacts.

Camera: If the user presses the camera button this will stop the currently executing program and start the camera application to take a new picture.

The way these buttons will be used has implications for the programs that we write. A program must be able to cope with the possibility that it will be removed from memory at any time, for example if the users decides to take a photograph while playing a game the game program will be stopped and may be removed from memory. Once the picture has been taken the user should be able to resume the game just it was left. The user should not notice that the game was stopped.

Programs that are about to be stopped are given a warning message and the Windows Phone operating system provides a number of ways that a program can store state information. We will explore how to do this later in the text.

Not all Windows Phone devices will have a physical keyboard for entering text but all devices will be able to use the touch screen for text entry.

Screen Capture

If you want to capture a screenshot of your Windows Phone application you can get one by pressing the Windows soft key and the power key simultaneously. The screen images are held in a folder called Screenshots in the Pictures part of the phone.

Memory and Storage

Memory is one of the things that computer owners have been known to brag about. Apparently the more memory a computer has the "better" it is. Memory actually comes in two flavors. There is the space in the computer itself where programs run and then there is the "mass storage" space that is used to store programs and data on the device.

On a desktop computer these are expressed as the amount of RAM (Random Access Memory) and the amount of hard disk space. A modern desktop computer will probably have around 4 gigabytes (four thousand megabytes) of RAM and around 500 gigabytes of hard disk storage. A megabyte is a million bytes (1,000,000). A gigabyte is a thousand million bytes (1,000,000,000). As a rough guide, a compressed music track uses around six megabytes, a high quality picture around three megabytes and an hour of good quality video will occupy around a gigabyte.

Windows 8 phones have at least 512 megabytes of RAM and at least 4GBytes of storage space. This means that a base specification Windows Phone will have an eighth the amount of memory and around a fiftieth of the amount of storage of a desktop machine. The Windows Phone operating system has been optimized to work in small amounts of memory. However, some of the things it does to make sure that the device doesn't run out of resources have an impact on how we write programs which we need to be aware of.

Network Connectivity

A mobile phone is actually the most connected device you can get. The one device has a range of networking facilities:

Wi-Fi: All Windows Phones support wireless networking. Wi-Fi provides a high speed networking connection, but only works if you are quite near to a network access point. Fortunately these are appearing all over the place now, with many fast food outlets and even city centers being Wi-Fi enabled.

3G: The next thing down from Wi-Fi in performance is the 3G (Third Generation) mobile phone network. The speed can approach Wi-Fi, but is much more variable. 3G access may also be restricted in terms of the amount of data a mobile device is allowed to transfer, and there may be charges to use this connectivity.

GPRS: In many areas 3G network coverage is not available. The phone will then fall back to the GPRS mobile network, which is much slower.

The networking abilities are exposed as TCP/IP connections (we will explore what this means later in the text).

Programmer's Point: Be able to handle changes of network state

Unfortunately, as you will have undoubtedly experienced, network coverage is not universal, and so software running on the phone must be able to keep going even when there is no data connection. Ideally the software should also cope with different connectivity speeds. There are some nice features in the emulation environment that allow you to simulate the effect of slow and intermittent network connection on your application, and you should certainly make use of these.

Near Field Communication (NFC)

Some Windows Phone 8 devices are fitted with Near Field Communication (NFC) technology. These phones contain an antenna which can power up and communicate with passive NFC "tags" which contain small amounts of data that can be read or written by software in the phone. There is great potential for novel applications using NFC technology. A phone can be triggered to perform an action when it "sees" a particular tag. Tags can be used to hold data about objects they are attached to. The data can be updated by software in the phone.

In addition two Windows Phone 8 devices can use NFC to perform "tap and send" communication. This allows the phones to exchange small amounts of data, for example to configure a network connection between them. We will investigate this technology later in the text.

Bluetooth

The Windows Phone also provides support for Bluetooth networking. This is used in the context of connecting external devices such as headsets and car audio systems, along with other devices. You can write programs for Windows Phone 8 devices that will allow them to interact with external devices using Bluetooth. You can also use Bluetooth to create a connection between two Windows Phones, for example for two player games.

Platform Challenges

The Windows Phone hardware is very powerful for a mobile device, but it is still constrained by the way that it musts be portable and battery power is limited. To make matters worse, users who are used to working with high speed devices with rich graphical user interfaces expect their phone to give them the same experience.

As software developers our job is to provide as much of that experience as possible, within the constraints of the environment provided by the phone. When we write programs for this device we need to make sure that they work in an efficient way that makes the most of the available platform. I actually see this as a good thing, I am never happy when programmers are allowed to get away with writing inefficient code just because the underlying hardware is very powerful.

Whenever we do something in we should consider the implications when running on a phone platform. I hope you will see that one effect of this is to make us all into much better programmers.

1.2 The Windows Phone Ecosystem

The Windows Phone is not designed as a device that stands by itself. It is actually part of an *ecosystem* which contains a number of other software systems that work around it to provide the user experience.

Connecting a Windows Phone to a Windows PC

All Windows Phones have a min-usb connector which can be used to link them to a PC. When a phone is first connected to the PC a device driver is automatically loaded which allows the user to browse the phone as they would any other storage device.

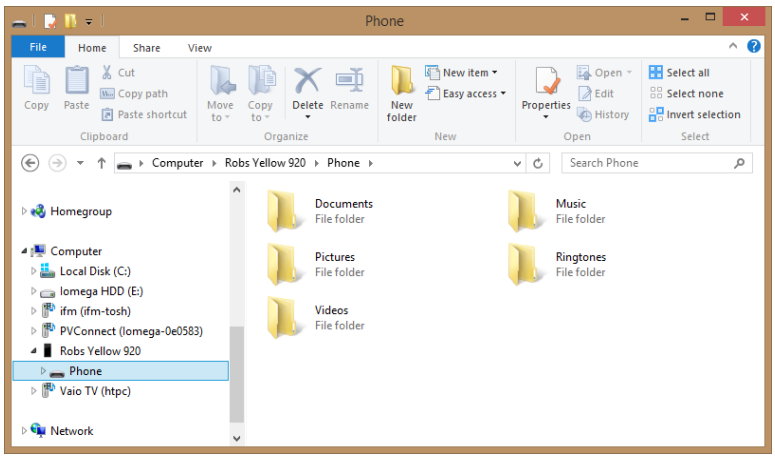


Figure 1-1 Navigating the Windows Phone file store

Users of a Windows Phone can drag files into and out of the folders on their device, to add music and documents and retrieve pictures taken with the phone.

In addition a Windows 8 application is available that makes it even easier to move applications between the phone and a PC. It also automatically copies photos from the phone onto the PC.

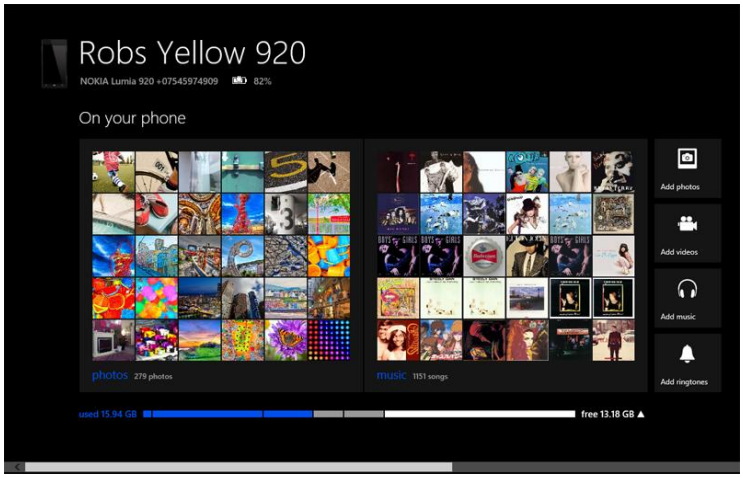


Figure 1-2 The Windows Phone Application

A Windows Phone 7 device is connected to a Windows PC by means of the Zune software, but this is not required for a Windows 8 Phone.

The programs that we write can make use of the media loaded onto the phone. It is very easy to write programs that load pictures or playback music and videos stored on the phone. It is also possible to write music playback programs that run in the background, behind other programs that the user may be running on the phone itself.

Windows Live and Xbox Live

The owner of a Windows Phone can register the phone to their Windows Live account. If their Windows Live account is linked to their Xbox Live gamer tag their gamer profile is imported onto their device and so they can participate in handheld gaming using the same identity as they use on their console.

A Windows Phone program can use the Windows Live identity of the owner and gamer tag information.

Bing Maps

We know that a phone contains hardware that allows it to be aware of its location. The Microsoft Bing maps service provides maps which cover most of the earth. A Windows Phone program can use its network connection to contact the Bing maps service and download maps and aerial views. There is a mapping control that makes it very easy for a program to add location displays. We will explore this in chapter 10.

Windows Notification Service

Although a phone has a wide range of networking technologies available there will still be occasions when it is unable to connect to a network. Windows Phone provides a notification service which allows programs to get notifications from the network even when the program is not active. The user is given a notification on the phone that they can use to start the application. Notifications are stored and managed by a notification service which will buffer them if the phone is not connected to the network when the notification is raised.

For example, you might have a sales system that needs to tell a customer when an item they have reserved is in stock. The ordering application on the phone can register with the notification service and then your stock management server can use the notification server to tell the customer when the item becomes available. This system can also be used in gaming, where one player wants to send a challenge to another.

Windows Phone and Windows Azure

Windows Azure is a set of "cloud" services that are made available by Microsoft. Rather than using your own computer to provide data storage and processing for internet users, you can instead rent space and processors in the cloud. If your networked application becomes very popular you don't have to buy more computers and connect them up, instead you just scale up your use of the cloud resource.

Server Applications

Cloud services are also very useful if you have to do a large amount of processing once, or very infrequently. Rather than having a large number of computers to satisfy the peak demands, instead you can just offload the work into the cloud. You can write the server applications in C# and test them on the cloud simulator before uploading them to do the work. Client applications can make use of these services via a variety of different network mechanisms.

Database Storage

While a Windows Phone can have a database inside the phone, it is often useful to use a remote database which is accessed via the network. This makes it easier to have really huge amounts of storage available, which you might want to update centrally. It is possible to create databases in the cloud and use them from a Windows Phone application.

BLOB Storage

Blob stands for *Binary Large Object*. It is perhaps my favourite name in this entire book. A blob can be anything, the text of a book, a movie or output from a medical scan. Windows Azure provides a storage facility where an application can store and retrieve blobs.

Authentication Services

Whenever a user wishes to get access to a resource there is always a problem of authentication. The resource provider will want to be sure that the user is who they say they are. This is a particular problem with portable (and steal able) devices such as phones. Writing high quality authentication software is very difficult, and not something you should undertake lightly. Windows Azure provides a way of taking the authentication service into the cloud.

The Windows Mobile Services

The Windows Azure web site provides project templates, class libraries and sample code that can get you started writing cloud applications using the phone. You can find it here:

```
http://www.windowsazure.com/en-us/develop/mobile/
```

Using the Ecosystem

It is important to remember that a phone is not just a phone these days. The connected nature of the application means that it will function in conjunction with remote services which are available over the network connection. Other parts of the phone make use of this connected ability too. The phone has a Facebook client built in and the camera application and picture storage elements can upload your pictures to Windows Live SkyDrive or Facebook. There are methods that make it easy to post updates to live feeds and even share favourites.

You can also use the network features of the phone to create your own client and server applications. You will need to write two programs to do this, one will run on the Windows Phone and the other can run on any computer connected to the internet, or even on a system in the "cloud".

1.3 Windows Phone program execution

The Windows phone provides a platform to run programs, some of which can be ones that we will write. It is worth spending some time considering how the programs are made to run on the phone and some of the implications for the way that we write them.

Application Switching on Windows Phone

Windows phone was designed on the basis that the user is king (or queen). All of the design decisions have been made to make sure that whatever the user is doing at that time, the phone is giving as much support to that activity as possible.

This means that when a user is running your application the phone is not running any other. If there are any other applications active on the phone they are "standing in the wings" waiting for their turn on stage.

Just like in a theatre show, where it is important that the performers can get on and off the stage quickly between scenes, it is important that it is easy for a user to switch between active applications. We have already seen that the Back button provides a quick way for a user to move from one program to another, when we create our applications we have to make sure the user has a good experience when they do this.

One implication for us developers is that we must live with the possibility that at any point our program might get "kicked off the stage" to make way for another. Then, if the user ever calls the application back from the wings, it should make sure that the show continues at exactly the same place.

The Windows phone operating system provides "Fast Application Switching" where an application is kept in memory while it waits in the wings for the call back on stage. In this situation the program has all its memory and resources intact and just needs to start running again. However, just like on a stage, where the space in the wings is limited, sometimes an application will be removed from memory (or sent back to its dressing room) if there is no longer room for it. The important point to remember though is that from the user's point of view the experience they get when they return to the application should be *exactly the same*. Just like it would take a performer longer to get from their dressing room back to the stage, an application that needs to be reloaded into memory will take a bit longer to return, but the experience the user gets when the application comes back should be exactly the same. A program is given a message when it is about to be moved "off stage" and it has access to storage that it can use to retain state information. We will see how to do this in Chapter 13.

Background Processing

The limitations imposed by the processor and battery life considerations make it hard for a device as small as a phone to run multiple processes. Windows PC users are familiar with a desktop cluttered with many different programs running simultaneously, but on a phone this is not practical, not least because the display is not large enough to show multiple applications.

The Windows Phone allows applications to contain "background tasks". A background task can take over when the main application is not able to run. They are allowed to run in carefully managed circumstances, so that if the phone has several background tasks active the user will not notice the phone running more slowly. Background tasks are designed for use in particular scenarios including background music playback, file transfer, regular updates and occasional bulk data processing. Background tasks are invisible to the phone user, although a user can see them running and control which ones are active. A background task can alert the user to an event by sending a notification or by updating the display on a "Live Tile" on the screen. The user can then start the application to find out more. We will see to create background tasks in Chapter 11.

Windows Phone and Managed Code

In the early days a computer ran a program just by, well, running a program. The file containing the program instructions was loaded into memory and then the computer just obeyed each one in turn. This worked OK, but this simple approach is not without its problems.

The first problem is that if you have a different kind of computer system you have to have a different file of instructions. Each computer manufacturer made hardware that understood a particular set of binary instructions, and it was not possible to take a program created for one hardware type and then run it on another.

The second problem is that if the instructions are stupid/dangerous the computer will obey them anyway. Stupid instructions in the file that is loaded into memory may cause the actual hardware in the computer to get stuck (a bit like asking me to do something in Chinese). Dangerous instructions could cause the program to damage other data in the computer (a bit like asking me to delete all the files on my computer).

The Microsoft Intermediate Language (MSIL)

Microsoft .NET addresses these problems by using an intermediate language to describe what a program wants to do. When a C# program is compiled the compiler will produce a file containing instructions in this intermediate language. When the program actually runs these instructions are compiled again, this time into the low level instructions that are understood by the target hardware in use. During the compilation process the instructions are checked to make sure they don't do anything stupid and when the program runs it is closely monitored to make sure that it doesn't do anything dangerous.

Microsoft .NET programs are made up of individual components called *assemblies*. An assembly contains MSIL code along with any data resources that the code needs, for example images, sounds and so on. An assembly can either be an executable (something you can run) or a library (something that provides resources for an application). A Windows Phone can run C# executable assemblies and libraries produced from any .NET compatible compiler. This means that you could write some of the library code in your application in another language, for example Visual Basic, C++ or F#. It also means that if you have libraries already written in these languages you can use them in phone applications.

The idea behind .NET was to provide a single framework for executing code which was independent of any particular computer hardware or programming language. The standards for .NET specify what the intermediate language looks like; the form of the data types stored in the system and also includes the designs of the languages C# and Visual Basic .NET.

Just in Time Compilation

When a program actually gets to run something has to convert the MSIL (which is independent of any particular computer hardware) into machine code instructions that the computer processor can actually execute. This compilation process is called *Just In Time* compilation because the actual machine code for the target device is compiled from the intermediate instructions just before it gets to run. The way in which the program runs within a monitored environment is called *managed code*. The compilation happens in the instant before the program runs, i.e. the user of the phone selects a program from the start page, and the MSIL is loaded from storage and then compiled at that time.

The downside of this approach is that rather than just run a file of instructions the computer now has to do more work to get things going. It must load the intermediate language, compile it into machine code and then keep an eye on what the program itself does as it runs. Fortunately modern processors (including the one inside the Windows Phone) can achieve this without slowing things down.

The upside is that the same intermediate code file can be loaded and executed on any machine that has the .NET system on it. You can (and I have) run exactly the same compiled program on a mobile device, Windows PC and Xbox 360, even though the machines have completely different operating systems and underlying hardware.

Another advantage is that this loading process can be expanded to ensure that programs are legitimate. .NET provides mechanisms by which programs can be "signed" using cryptographic techniques that make it very difficult for naughty people to modify the program or create "fake" versions of your code. If you become a Windows Phone developer you will be assigned your own cryptographic key that will be used to sign any applications that you create.

Finally, the use of an intermediate language means that we can use a wide range of programming languages. Although the programming tools for Windows Phone are focused on Visual Basic and C# it is possible to use compiled code from any programming language which has a .NET compiler. If you have existing programs in C++ or even F# you can use intermediate code libraries from these programs in your Windows Phone solutions. Remember though that you will need to use C# for the "front end" for these programs.

Managed Code

Programs that you create on Windows Phone run within the "managed" environment provided by the operating system. You will write your C# programs and then they will run inside a safe area in the phone provided by the operating system. Your program will not be allowed direct access to the hardware. This is good news because it stops people from writing programs that stop the proper operation of the phone itself. As far as the developer is concerned this actually makes no difference. When a program calls a method to put a picture on the screen the underlying systems will cause the picture to appear.

Developer Implications

As far as a developer is concerned there is an element of good news/bad news about all this. The good news is that you only need to learn one programming language (C#) to develop on a range of platforms. Programs that you write will be isolated from potential damage by other programs on a system and you can be sure that software you sell can't be easily tampered with.

The bad news is that all this comes at the expense of extra work for the system that runs your application. Starting up a program is not just a case of loading it into memory and obeying the instructions it contains. The program must be checked to ensure that it has not been tampered with and then "just in time" compiled into the instructions that will really do the work. The result of all this is that the user might have to wait a little while between selecting the application and getting the first screen displayed.

Fortunately the power of Windows Phone means that these delays are not usually a problem but it does mean that if we have very large programs we might want to break them into smaller chunks, so that not everything is loaded at once. But then again, as sensible developers we would probably want to do this kind of thing anyway.

Windows Phone and Compiled Code with C++

While managed code helps us make programs that are portable and well-behaved the overheads of the compilation process and the run time environment do have an impact on performance. To allow programmers to get the very last bit of performance out of the Windows Phone process it is possible to create programs using the C++ language. These programs are compiled directly into machine code that runs at full speed in the device. This is useful when you want a program to run as quickly as possible, for example in the case of a game, but for the great majority of applications there is no need to sacrifice the convenience and ease of use provided by managed code to get this extra speed.

Most of the time our programs will be waiting for the user to provide input or for data to arrive from the network, and in these situations the C# language and the .NET framework provide ample performance. For this reason we will not be exploring the C++ development option in this text.

The MonoGame Framework and XNA

Earlier versions of the Windows Phone platform allowed programmers to create games using the XNA Framework. You can still create XNA games using Visual Studio 2012 and these will run on Windows 7 and Windows 8 devices. However these games will not be able to take advantage of the additional features in Windows Phone 8.

If you want to write XNA games that can run as Windows Phone 8 applications you can use the MonoGame framework. It is a very good idea to use MonoGame for your game development as versions of this XNA implementation are available for the Windows 8, Android and IOS operating systems. It is therefore quite simple to take a single game and deploy it across a wide range of devices.

You can find out more about game development using MonoGame in Chapter 8.

1.4 Windows Phone application development

You write Windows Phone applications in exactly the same way as you write other applications for the Windows desktop. You use the Visual Studio IDE (Integrated Development Environment). You can debug a program running in a Windows Phone device just as easily as you can debug a program on your PC desktop.

You can take all your Windows Desktop development skills in Silverlight or Windows Presentation Foundation (WPF). If you learn how to use the Windows Phone you are also learning how to write code for the desktop (Silverlight). This is great news for you as it means that you can write (and even sell) programs for Windows Phone without having to learn a lot of new stuff. If you have previously written programs for desktop computers then the move to Windows Phone development will be a lot less painful than you might expect.

Visual Studio 2012 and Windows Phone

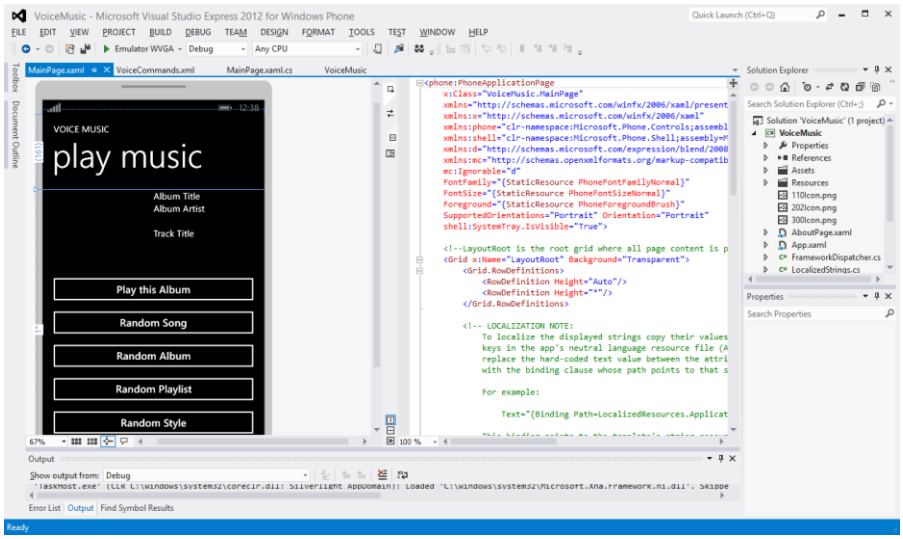


Figure 1-3 Work in Progress using Visual Studio 2012

Visual Studio provides a very powerful environment where you can create, build and test your Windows Phone applications. Figure 1-3 Work in Progress using Visual Studio 2012 shows an application under development. On the far

left you can see a design surface where the programmer can create the look and feel of the application. The application user interface is expressed using a mark-up language which you can see in the centre. Elements in the display can be connected to program components which can be written in C# or Visual Basic.

On the far right of Figure 1-3 you can see the Solution Explorer, which makes it easy for a developer to manage the different assets and code that make up the entire solution. These will be packaged up by Visual Studio into a single file which ends up being loaded into the Windows Phones of users who download your program from the Windows Phone Store.

You can get Visual Studio 2012 Express Edition from the Windows Phone Developer website:

```
http://dev.windowsphone.com
```

It is a free download and provides everything you need to create Windows Phone applications. You can install Visual Studio 2012 Express Edition directly from the web. This can take a while to install, but the process is largely automatic, you just have to watch the progress bars.

The Windows Phone Emulator

The Windows Phone development environment is supplied with an emulator which gives you a Windows Phone you can play with on your PC desktop. If you have a PC system that supports multi-touch input you can even use this with the emulator to test the use of multi-touch gestures with your Windows Phone programs.

The emulator is loaded with all the Windows Phone applications that you get with a real device. You can use it to surf the internet using the Windows Phone browser. This makes it a really useful way of testing how your web sites will look on a mobile device.

Figure 1-4 The Windows Phone emulator

Figure 1-4 shows the emulator screen, complete with email, calendar, people and pictures applications. While the emulator is feature complete, in that it behaves exactly like a real phone would in response to the requests from your software, it does not mimic the performance of the actual phone hardware.

Programs running on the emulator are using the power of your PC, which may well be much greater than the processor in the phone. This means that although you can test the functionality of your programs using the emulator you only really get a feel for how fast the program runs, and what the user experience feels like, when you run your program on a real device.

The emulator will allow you to test motion controlled programs by allowing you to tip and turn a "virtual" phone on your PC screen. The inputs to the program will reflect the orientation that you see on your screen. You can also pre-record a set of movements to allow you to perform repeatable tests using particular movements or gestures.

There is provision for location emulation. You can select on a map where you want the emulated phone to "be". You can also input paths for the phone to follow and then replay these, so that you can create journeys to test location aware applications.

You can also simulate the effects of poor network connectivity, to test how your application behaves when it loses touch with the network.

The Windows Phone Emulator and "Hyper-V" Threading

Earlier versions of the Windows Phone emulator ran as processes on your PC. This means that while they looked like a separate device, they were actually just a program pretending to be that separate device. The Windows Phone 8 Emulator makes use of hardware and operating system features that allow it to run in a virtual computer, alongside the one running your Windows 8 environment. This is good news, because it means that the simulation is much more realistic. However, it is also bad news because your PC needs to support the hardware and software environment that the emulator runs in. This means having a computer processor with "Second Level Address Translation" or (SLAT) capability and the Professional version of the Windows 8 operating system. The install process for Visual Studio 2012 Express edition, which we will be looking at later, will check for these and also enable "Hyper-V" threading, which is required to enable a single computer system to support these multiple processor personalities.

Accessing Windows Phone Facilities

The Windows Phone platform provides a library of software resources that let your programs make use of the features provided by the device itself. Your programs can make use of the camera in the phone, place phone calls and even send SMS messages. They can also make use of the GPS resources of the phone to allow you to create location aware applications. The facilities are divided into *Launchers* which let your program transfer to another application and *Choosers* which use the system to select an item and then return control to your program. We will see how to use these later in this text.

Windows Phone Connectivity

As you might expect, programs on a Windows Phone are extremely well connected. Applications on the phone can make use of the TCP/IP protocol to connect to servers on the internet. Programs can use Web Services and also set up REST based sessions with hosts. If you are not sure what any of this means, don't worry. We will be exploring the use of network services later in the text.

Data Storage on Windows Phone

Useful applications need to store data. Some of this data will be simple things, for example configuration data such as the size of the display. Other items will be larger; perhaps high scores or the place the player has reached in the game. Then there may be a need for even more information that must be held in a structured way, for example a database of customers or products. The Windows Phone provides appropriate storage at all these levels.

Isolated Storage

Isolated storage is so named because the storage space for one application is isolated from all the others. When a program is installed on the Windows Phone it is given access to a storage space that is completely isolated from all others. There is storage that can provide simple name/value pairs (for example ColourScheme:Purple) for things like program settings. Applications can also create complete file hierarchies within their patch of isolated storage, depending on their needs.

A program called the *Isolated Storage Explorer* can be used to view the folders and files in isolated storage when you are developing an application so that you can make sure they contain the correct data.

Local Database

A database is a collection of data which is managed by a program that is also, rather confusingly, called a database. Databases are used to create programs that store and manipulate huge amounts of data and make it very easy for our applications to provide answers to questions like "Tell me which of our customers from New York bought a pair of shoes in November". The Windows Phone has a database built-in. This makes it very easy to create an application that stores structured data.

Each Windows Phone application also has access to a single SQL database where they can store very large amounts of structured data. An application can create queries using the LINQ (Language INtegrated Query) libraries to extract and modify data in the database. The actual database file is held in the Isolated Storage for the application. We will see how to use databases and LINQ in Chapter 6.

Development Tools

The tools that you need to get started are free. You can download a copy of the Windows Phone SDK and be started writing Windows Phone applications literally within minutes. Developers who have more advanced, paid for, copies of Visual Studio 2010 can use all the extra features of their versions in mobile device development by adding the Windows Phone SDK plugin to their systems. You can download a copy of the Windows Phone SDK from:

```
http://dev.windowsphone.com.
```

Performance Profiling

Applications for a Windows PC are not usually constrained by the cpu performance. The sheer power of the processors in in desktops and laptops means that quite inefficient code will still be useable. However, for the Windows Phone it is important to make our programs as speedy and efficient as possible, both to give the best user experience and also to make sure that our programs don't use too much battery power.

The Windows Phone SDK includes a performance profiler that we can use to view the demands our applications place on the processor, and also find out where programs are spending most of their time. This makes it possible to determine which methods in a program should be optimised to get the best improvements.

Windows Phone Store

Windows Phone Store is where you can sell programs that you have created. It is moderated and managed by Microsoft to ensure that applications meet certain minimum standards of behaviour. If you want to sell an application you must be a registered developer and submit your program to an approvals process.

Developer Registration

It costs $99 to register as a developer, but students can register for free via the Dreamspark programme. Developers can register their Windows Phone devices as "development" devices. Visual Studio can deploy applications to a developer device and you can even step through programs running in the device itself.

Distributing Applications

You can distribute free applications as well as paid ones. A registered developer can submit up to 100 free applications for approval in any year. You can also create applications that have a "demo" mode which can be converted into a "full" version when the user purchases them. Your program can tell when it runs whether it is operating in full or demo mode.

When you submit an application to be sold in the store it will go through an approvals process to make sure that it doesn't do anything naughty and that it works in a way that users expect. If the approvals process fails you will be given a diagnostic report. Before you send an application into the system you should read the certification guidelines of the store.

Testing Tool

The Windows Phone SDK contains a testing tool which can be used to pre-test an application before it is submitted to the store. This tool automatically synchronises with the approval process, so if the process changes the tool will be updated to match. Some of the testing is automatic, but the tool also describes each of the usage tests that will be performed by the testing team. This greatly improves the chances of us making an application that will be approved first time.

Unlocking your own phone

If you are not a registered developer you can still create and deploy applications to your own Windows Phone device. All you need is a Microsoft Account (which is free) to be able to unlock your phone for developer access and deploy two of your programs onto it. This is really useful, because you can use Visual Studio to run and debug your programs inside the device.

Private Betas

One great way to find out the quality of an application or game is to give it to a bunch of people and see what they say. We can create "invitation only" releases of a product so that up to 100 people can be sent a download link for our new program. This allows the testers up to 90 days of use of the program, during which time they can give us their feedback.

What We Have Learned

1. Windows Phone 8 is a powerful computing platform, built using the same operating system foundations as Windows 8 Desktop.
2. The Windows 7 and 7.5 phone platforms have now been superseded by Windows 8
3. All Windows Phone devices have a core specification. This includes a particular size of display, capacitive touch input that can track at least four points, Global Positioning System support, 3D graphics acceleration, high resolution camera and ample memory for program and data storage.
4. The Windows Phone device can be connected to a Windows PC to transfer information to and from the phone. This can be performed using the Windows Explorer, or via a Windows 8 application.
5. Windows Phone systems can make use of network based services to receive notifications, determine their position and perform searches.
6. The Windows Phone operating system supports full multi-tasking, but to for the best performance only one user application can be active at one time. However, applications can create their own "background agents" that can perform specific tasks when the application is not running. Windows Phone also provides "Fast Application Switching" that retains applications in memory so that they can be resumed very quickly.
7. The Windows Phone runs programs that have been compiled into Microsoft Intermediate Language (MSIL). This MSIL is compiled inside the phone just before the program runs. The programs themselves run in a "managed" environment which stops them from interfering with the operating of the phone itself. It is also possible to run programs that have been compiled to run directly on the Windows Phone hardware when you are determined to get the best possible performance from the device.
8. When developing software for Windows Phone you can create C# applications that use XAML to define the display behaviour using Visual Studio 2012. Programmers can use a Windows Phone emulator that runs on Windows PC and provides a simulation of the Windows Phone environment.
9. Games can be written in as Windows Phone applications, but you can also use the MonoGame environment to create games using the XNA framework. Games created using MonoGame can also be ported onto a number of other platforms, including Windows 8, Android and iOS (Apple).
10. Programs have access to all the phone facilities and can place calls, send SMS messages etc.
11. The Windows Phone SDK can be used to create Windows Phone applications. It is a free download from `dev.windowsphone.com` However, to deploy applications to a phone device you must be a registered Windows Phone Developer. This costs $99 per year but registration is free to students via the Microsoft Dreamspark initiative. A registered developer can upload their applications to Windows Phone Store for sale.

2 Making a User Interface with XAML

A user interface is a posh name for the thing that people actually see when they use your program. It comprises the buttons, text fields, labels and pictures that the user actually works with to get their job done. Part of the job of the programmer is to create this "front end" and then put the appropriate behaviours behind the screen display to allow the user to drive the program and get what they want from it. In this section we are going to find out about XAML, and how to use it to create a user interface for our programs.

2.1 Program Design with XAML

The *User Interface* for an application is an incredibly important element. It is the thing that the user will see when the application begins running, and it is the thing that they will work with. If the user interface is hard to understand, slow or perhaps just the wrong colour, then users may well reject your solution in preference to another.

It turns out that most programmers are not that good at designing attractive user interfaces (although I'm sure that you are). In real life a company may employ graphic designers who will create artistic looking front ends. The role of the programmer will then be to put the code behind these displays to get the required behaviours in the program. XAML recognises this process by enforcing a very strong separation between the user interface design and the code that is controlled by it. This makes it easy for a programmer to create an initial user interface which is subsequently changed by a designer into a much more attractive one. It is also possible for a programmer to take a complete user interface design and then fit the behaviours behind each of the display components.

XAML

You are probably wondering what XAML is. It sounds rather like a planet that some intrepid space explorer might have to invade at some point, but actually it is an abbreviation of "eXtensible Application Markup Language". To take each word at a time:

- **eXtensible**: users expect to have new and interesting ways to interact with their computers, so any language we use to describe a user interface, (i.e. the things that he user sees on the screen) must be capable of extending to incorporate these
- **Application**: this is a posh name for a program or game that somebody might like to use
- **Markup**: a markup language is way that you can add meaning to some data. A composer would write things like "Fortissimo" on his music when she wanted something played really loudly. The letter M in the HTML standard used for web pages also stands for "markup". In the case of HTML the markup lets the page designer specify that text should be bold, or underlined, or in a particular font. In the case of XAML the markup lets the designer describe the buttons, images, and text labels that are to be drawn on the screen
- **Language**: when we write English we have to observe certain conventions, such as starting sentences with a capital letter and ending them with a full stop. These standards are laid down in the rules of grammar for English. XAML also has a grammar, to make sure that computers and people can make proper use of the designs expressed using it.

Every page of a Windows Phone application will be described by a XAML file that sets out exactly where all the display elements are and how they behave. For example, a button might slide onto the page, or flash when you tap it. These behaviours will all be expressed using the XAML description.

We have already seen that Visual Studio provides a place where a programmer can design the user interface for an application. However, for the finer points of a display design, for example animations, you need a graphical design tool, which is where Blend comes in.

Using Blend for Visual Studio to create XAML

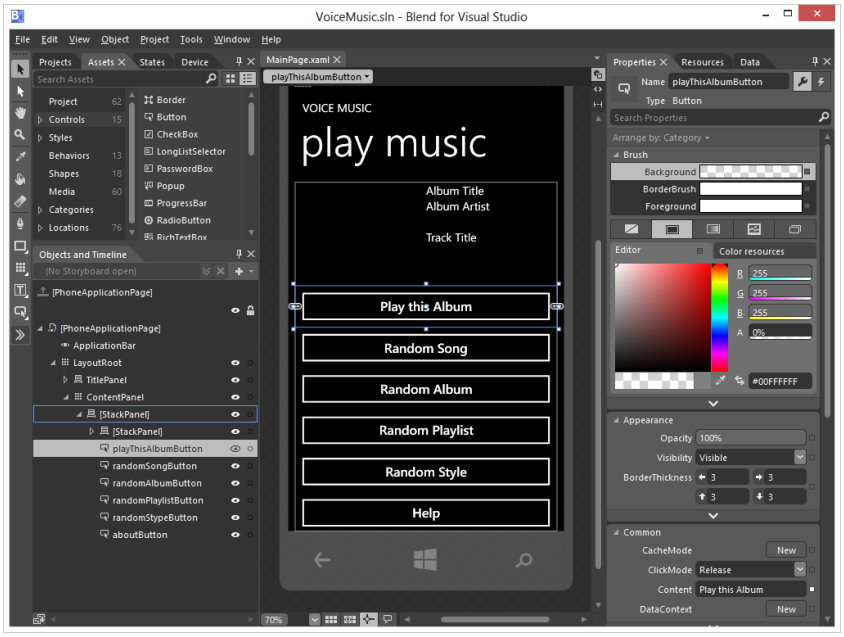


Figure 2-1Blend for Visual Studio

Figure 2-1 shows how Blend displays the work in progress that we saw back in Figure . The controls in Blend let the designer focus on colour, transparency (how much of the background shows through) and animation.

The idea is that the user interface designer uses the Expression Blend tool and the programmer uses the Visual Studio tool to create the code. However it is important that you remember that the designs are still expressed in XAML form which ever tool is used to create them. A "rose tinted pink button that slides in from the left and then slows down in the centre of the screen and changes colour to yellow" would be expressed as a lump of text containing a set of values that give the speed, position and transitions that the button would go through.

We won't be spending much time learning about Blend, it is a specialised tool that has been created for designers, not developers. However, it can be fun to explore, and it is very easy to use Blend to open a XAML page design from within Visual Studio.

The Windows Phone SDK (Software Development Kit) includes versions of Visual Studio and Blend that are installed together.

The Windows Phone Design Style

From a design point of view a nice thing about Windows Phone is that it brings with it a whole set of design guidelines. These guidelines set out how controls are supposed to look and establish a set of criteria that your applications should meet if they are to be "good looking" Windows Phone applications. This style regime is carried over into the built in components that are provided for you to use in your programs. The happy consequence of this is that if you use the Buttons, TextFields and other components that are supplied with Visual Studio you will be automatically adhering to the style guidelines.

This is a great help those of us who are not very good at design because it means that we are only ever going to use things that look right. Of course you can completely override the properties of the supplied components if you really want purple text on an orange background but I would not really advise this.

There is actually a style resource you can use if you really want to know how to make "proper looking" programs. This is well worth a look if you want to publish your programs. Users will have certain expectations of how things should look and work. You can find the style guide here:

`http://msdn.microsoft.com/en-us/library/hh202915.aspx`

For the purpose of this course we are going to use the XAML design tools in Visual Studio. These do not give all the graphical richness that you can achieve with Blend, but for programmers they are more than adequate. This should make sure that our applications adhere to the guidelines and are clean and simple to use.

One piece of good news is that if you learn how to use the XAML user interface design process on Windows Phone you will not have a problem moving to the Windows 8 desktop and tablet environments, as these work in exactly the same way

XAML Elements and Software Objects

From a programming point of view each of the elements on the screen of a display is actually a software object. We have come across these before. A software object is a lump of behaviours and data. If we were creating an application to look after bank accounts we could create a class to hold account information:

```
public class Account
{
  private decimal balance ;
  private string name ;

  public string GetName ()
  {
     return name;
  }


  public bool SetName (string newName){
  {
    // Final version will validate the name
    name = newName;
    return true;
  }

  // Other get and set methods here

}
```

This class holds the amount of money the account holder has (in the data member called `balance`) and the name of the account holder (in the data member called `name`). These can be called *properties* of the object.

If I want to make a new `Account` instance I can use the new keyword:

```
Account rob = new Account();
rob.SetName("Rob");
```

This makes a new `Account` instance which is referred to by the reference `rob`. It then sets the name member of this `Account` to "`Rob`". We are used to creating objects which match things we want to store. When we make games we could invent a `Sprite` class to hold the picture of the sprite on the screen, the position of the sprite and other information.

Objects are a great way to represent things we want to work with. It turns out that objects are also great for representing other things too, such as items on a display. If you think about it, a box displaying text on a screen will have properties such as the position on the screen, the colour of the text, the text itself and so on. Consider the following:

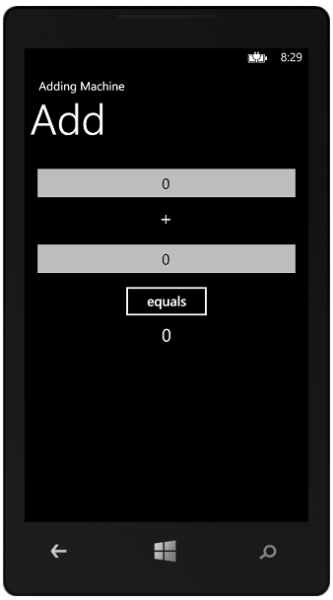


Figure 2-2 Simple Adding Machine

Figure 2-2 shows a very simple Windows Phone program that I've called an "Adding Machine". You can use to perform very simple sums. You just enter two numbers into the two text boxes at the top and then press the equals button. The program then rewards you with the sum of these two numbers. At the moment it is showing us that 0 plus 0 is 0. Each individual item on the screen is called a *UIElement* or User Interface element. I'm going to call these things elements from now on. There are seven of them on the screen above:

1. The small title "Adding Machine". This is known in the Windows Phone style guidelines as the 'Application Title'.
2. The larger title "Add". This is known as the 'Page Title'.
3. The top textbox, where I can type a number.
4. A text item holding the character +.
5. The bottom textbox, where I can type another number.
6. A button, which we press to perform the sum.
7. A result textbox, which changes to show the result when the button is pressed.

Each of these items has a particular position on the screen, particular size of text and lots of other properties too. We can change the colour of the text in a text box, whether it is aligned to the left, right or centre of the enclosing box and lots of other things too.

There are three different types of element on the screen:

1. `TextBox` – allows the user to enter text into the program.
2. `TextBlock` – a block of text that just conveys information.
3. `Button` – something we can press to cause events in our program.

If you think about it, you can break the properties of each of these elements down into two kinds, those that all the elements need to have, for example position on the screen, and those that are specific to that type of element. For example only a `TextBox` needs to record the position of the cursor where text is being entered. From a software design point of view this is a really good use for a class hierarchy.

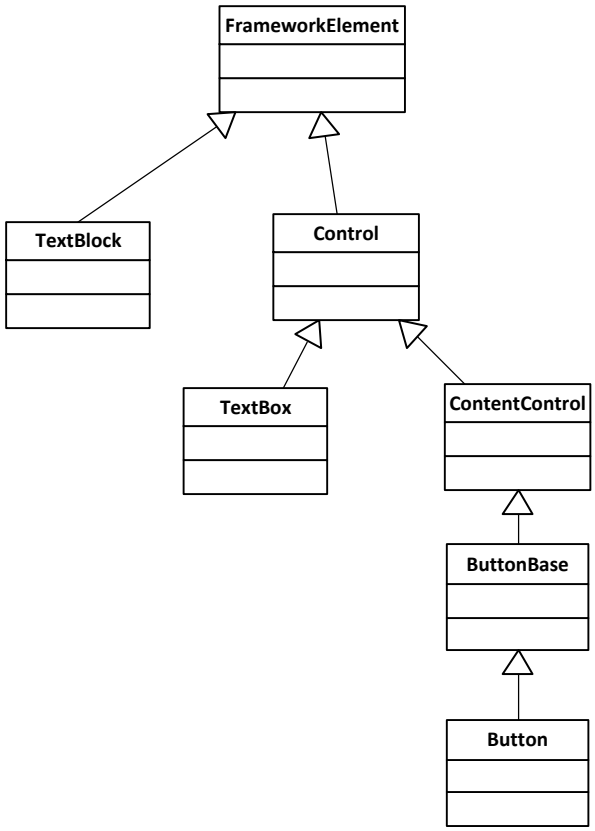


Figure 2-3 Part of the XAML class hierarchy

Figure 2-3 above shows part of the hierarchy that the XAML designers built. The top class is called `FrameworkElement`. It contains all the information that is common to all controls on the screen. Each of the other classes is a child of this class. Children pick up all the behaviours and properties of their parent class and then add some of their own. Actually the design is a bit more complex than shown above, the `FrameworkElement` class is a child of a class called `UIElement`, but it shows the principles behind the controls.

Creating a class hierarchy like this has a lot of advantages. If we want a custom kind of textbox we can extend the `TextBox` class and add the properties and behaviours that we need. As far as the XAML system is concerned it can treat all the controls the same and then ask each control to draw itself in a manner appropriate to that component.

So, remember that when we are adjusting things on a display page and creating and manipulating controls we are really just changing the properties of objects, just as we would change the name and balance values of a bank account object. When we design a XAML user interface we set the data inside the display elements to position them on the display. Next we are going to find out how to do this.

The Toolbox and Design Surface

We could start by investigating how the Adding Machine above was created. It turns out to be really easy (we will take a really detailed look at Visual Studio in the next section). When we create a brand new Windows Phone project we get an empty page and we can open up a ToolBox which contains all the controls that we might want to add to the page:

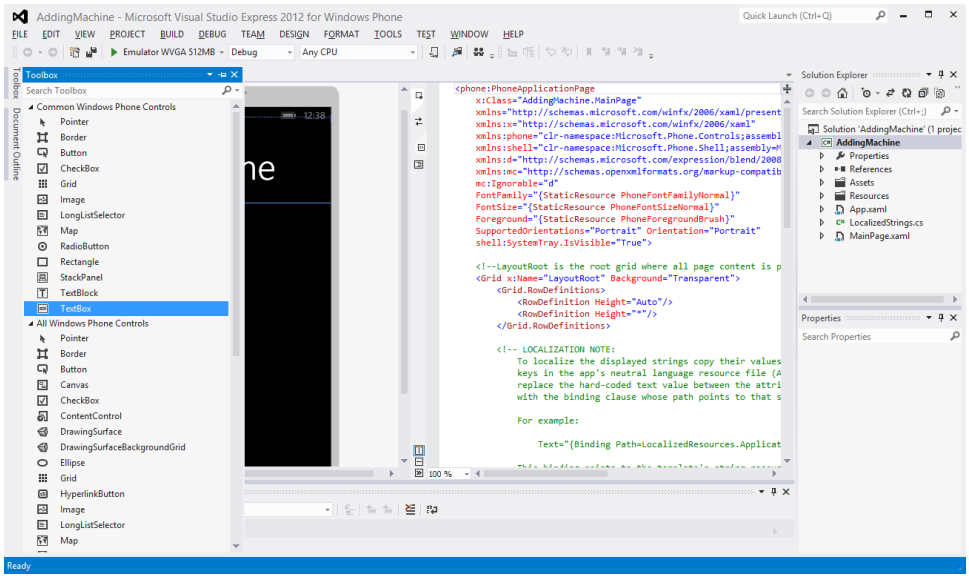


Figure 2-4 Visual Studio Toolbox

Figure 2-4 shows the ToolBox which I have opened up over the Windows Phone screen design surface. You can also see the text of the XAML that describes this page in the middle of the Visual Studio display. If we use the Toolbox to add an item to the page the XAML will be updated with the text description of the item that we added.

We can assemble the user interface by simply dragging the controls out of the toolbox onto the design surface.

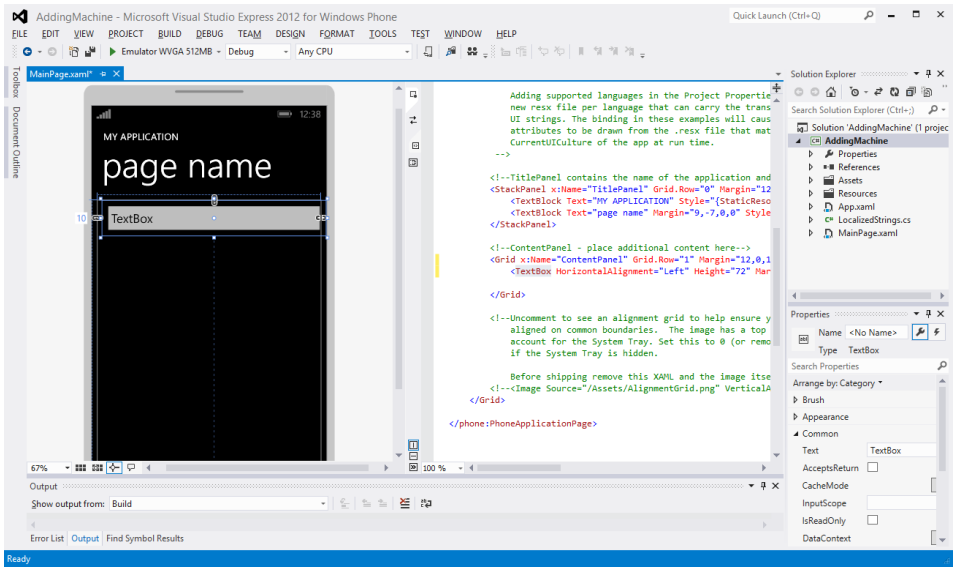


Figure 2-5 Adding a TextBox

Figure 2-5 above shows the effect of dragging a `TextBox` out of the Toolbox area and dropping it onto the Windows Phone page. A `TextBox` is a display element that allows the user of the program to enter text into the program. If they select this area of the screen a keyboard is displayed which can be used to edit the text in the box.

Note that in the XAML, right in the middle of the page, you can see some text that describes a `TextBox` with a HorizontalAligmnent set to “Left” and a Height of 72.

```
<TextBox HorizontalAlignment="Left" Height="72" Margin="0,20,0,0" TextWrapping="Wrap"
Text="TextBox" VerticalAlignment="Top" Width="456"/>
```

If we run the program now we can see the `TextBox` on the screen:

Figure 2-6 Displaying a Textbox

Figure 2-6 shows the Windows Phone emulator running our program, which is showing the textbox we added to the display. You may be wondering about the meaning of the strange little numbers down the right hand side of the phone screen. These are performance counters which are used during development to find out the demands that a program is making on the hardware. We will see how to remove these later.

If you have ever done any Windows Forms or XAML development (perhaps using earlier versions of .NET) then this will all be very familiar to you. If you haven't then it is a quick and easy way to build up a user interface. Because all the controls are already styled correctly, just like the Windows Phone user interface, you are automatically building an application that looks like a "real" one. The designer has lots of nice features that make it very easy to align your controls too, which is nice.

Managing Element Names in Visual Studio

Once we have put some elements on the screen we need to set the names of them to sensible values. The designer forces us to set the names of the elements, if you look at the XAML above you will find that there is no name property for the TextBox that we just created. This is actually quite sensible, as a program may never have to refer to some things on the screen, for example labels and prompts. However, at some point a program will want to refer to the `TextBox` that we have created, and so we need to give it a name. We can change the properties of things on the screen by clicking on the item in the designer and then seeking out the Properties pane in Visual Studio for that item.

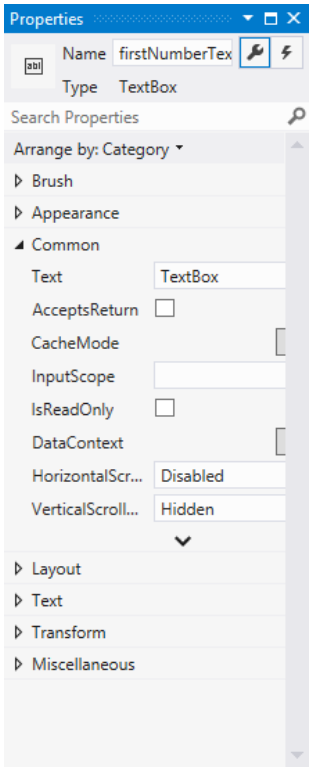


Figure 2-7TextBox properties

Figure 2-7 above shows the properties window for the top `TextBox` on the page. The name of an element is given at the very top of the window. As we saw above, initially an item has no name. You add a name when you want to refer to that object from within your program.

I have set the name of this textbox to `firstNumberTextBox`. You'll never guess what the second text box is called. Note that the name of a property in this context is actually going to set the name of the variable declared inside the adding machine program. In other words, as a result of what I've done above there will now be the following statement in my program somewhere:

```
TextBox firstNumberTextBox;
```

Visual Studio looks after the declaration of the actual C# variables that represent the display elements that we create and so we don't need to actually worry about where the above statement is. We just have to remember that this is how the program works.

Properties in XAML Elements

Once we have given our TextBox display element a proper name we can move on to give it all the properties that are required for this application. We can also change lots of properties for a TextBox including the width of the TextBox, the margin (which sets the position on the screen) and so on. The values that you see in the properties windows above are ones that reflect the current position and size of the item on the screen. If I drag the item around the margin values will change. If I change the values in the window above I will see the item move in the design surface. Again, if you are a Windows Forms kind of person you will see nothing strange here. But you must remember that all we are doing is changing the content of an object. The content of the properties window will change depending on what item you select

XAML element properties and C# properties

When we talk about the "properties" of XAML elements on the page (for example the text displayed in a `TextBox`) we are actually talking about property values in the class that implements the `TextBox`. In other words, when a program contains a statement such as:

```
resultTextBlock.Text = "0";
```

- this will cause a `Set` method to run inside the `resultTextBlock` object which sets the text on the `TextBlock` to the appropriate value. At this point it is worth refreshing our understanding of properties in C# so that can get a good understanding of what is happening here.

C# classes and properties

C# classes can contain member data (the name of the person owning a particular bank account) and behaviours (methods called `GetName` and `SetName` that let a program find out this name and change it). Properties provide a way of representing data inside a class that is easier to work with than using `Get` and `Set` methods. They are used extensively in managing XAML elements, and therefore worth a refresher at this point.

A property is a member of a class that holds a value. We have seen that we can use a member variable to do this kind of thing, but we need to make the member value public. For example, the bank may ask us to keep track of staff members. One of the items that they may want to hold is the age of the bank account holder. We could do it like this:

```
public class Account
{
   public int Age;
}
```

The class contains a public member. I can get hold of this member in the usual way:

```
Account s = new Account ();
s.Age = 21;
```

I can access a public member of a class directly; just by giving the name of the member. The problem is that we have already decided that this is a bad way to manage our objects. There is nothing to stop things like:

```
s.Age = -100210232;
```

This is very naughty, but because the `Age` member is `public` we cannot stop it.

Creating Get and Set methods

To get control and do useful things we can create get and set methods which are `public`. These provide access to the member in a managed way. We then make the `Age` member `private` and nobody can tamper with it:

```
public class Account
{
   private int age;
   public int GetAge()
   {
      return this.age;
   }
   public void SetAge( int inAge )
   {
      if ( (inAge > 0) && (inAge < 120) )
      {
         this.age = inAge;
      }
   }
}
```

We now have complete control over our property, but we have had to write lots of extra code. Programmers who want to work with the age value now have to call methods:

```
Account s = new Account ();
s.SetAge(21);
Console.WriteLine ( "Age is : " + s.GetAge() );
```

Using Properties

Properties are a way of making the management of data like this slightly easier. An age property for the `StaffMember` class would be created as follows:

```
public class Account
{
   private int ageValue;

   public int Age
   {
      set
      {
         if ( (value > 0) && (value < 120) )
            ageValue = value;
      }
      get
      {
         return ageValue;
      }
   }
}
```

The age value has now been created as a property. Note how there are get and set parts to the property. These equate directly to the bodies of the get and set methods that I wrote earlier. The really nice thing about properties is that they are used just as the class member was:

```
Account s = new Account ();
s.Age = 21;
Console.WriteLine ( "Age is : " + s.Age );
```

When the `Age` property is given a value the `set` code is run. The keyword `value` means "the thing that is being assigned". When the `Age` property is being read the `get` code is run. This gives us all the advantages of the methods, but they are much easier to use and create.

Data validation in properties

If we set an age value to an invalid one (for example we try to set the age to 150) the set behaviour above will perform validation and reject this value (nobody over 120 is allowed to have an account with our bank), leaving the age as it was before. A program that is using our Age property has no way of knowing whether an assignment to the property has failed unless it actually checks that the assignment worked.

```
Account s = new Account ();
int newAge = 150;
s.Age = newAge;
if (s.Age != newAge )
    Console.WriteLine ( "Age was not set" );
```

The code above sets the age to 150, which is not allowed. However, the code then tests to see if the value has been set. In other words, it is up to the users of your properties to make sure that values have been set correctly. In the case of a Set method the method itself could return false to indicate that the set has failed, here the user of the property has to do a bit more work.

Multiple ways of reading a property

Properties let us do other clever things too:

```
public int AgeInMonths
{
   get
   {
      return this.ageValue*12;
   }
}
```

This is a new property, called `AgeInMonths`. It can only be read, since it does not provide a set behaviour. However, it returns the age in months, based on the same source value as was used by the other property. This means that you can provide several different ways of recovering the same value. You can also provide read-only properties by leaving out the set behaviour. Write only properties are also possible if you leave out the get.

Properties and notifications

You may be asking the question "Why do we use properties in XAML elements?" It makes sense to use them in a bank account where I want to be able to protect the data inside my objects but in a program, where I can put any text I like inside a `TextBlock`, there seems to be no point in having validation of the incoming value. In fact running this code will slow down the setting process. So, by making the `Text` value a public string we could make the program smaller and faster, which has to be a good thing. Right?

Well, sort of. Except that when we change the text on a `TextBlock` we would like the text on the page in front of the user to change as well. This is how our adding machine will display the result. If a program just changed the value of a data member there would be no way the display system could know that the message on the screen needs to be updated.

However, if the `Text` member is a property when a program updates it the set behaviour in the `TextBlock` will get to run. The code in the set behaviour can update the stored value of the text field and it can also trigger an update of the display to make the new value visible. Properties provide a means by which an object can get control when a value inside the object is being changed, and this is extremely important. The simple statement:

```
resultTextBlock.Text = "0";
```

- may cause many hundreds of C# statements to run as the new value is stored in the TextBlock and an update of the display is triggered to make the screen change.

Page Design with XAML

We can complete the page of our Adding machine by dragging more elements onto the screen and setting their properties. Remember that the first thing I do after dragging an item onto the page is set the name of that element. One of the first symptoms of a badly written program (for me) is a page that contains lots of elements called "Button1" and "Button2". In fact, this must have bothered the designers of XAML too, as when you create a new element by dragging it onto the design surface it initially has no name. You have to put the name in yourself, and hopefully you will think of a sensible one.

From the sequence above I hope you can see that designing a page looks quite easy. You just have to drag the elements onto the page and then set them up by using the properties of each one. If you read up on the XAML language you will find that you can give elements graphical properties that can make them transparent, add images to their backgrounds and even animate them around the screen. These are the properties that can be manipulated using the Blend tool. At this point we have moved well beyond programming and entered the realm of graphic design. And I wish you the best of luck.

2.2 Understanding XAML

In the above section we saw that a user interface design can be broken down into some element manipulation using the Visual Studio designer and some property setting on the elements that you create. Once you know how to do these things, you can create every user interface you need.

It is perfectly true that you can work this way, and you could even think about making every program you ever need without touching XAML, although I think you would be missing out on some of the most powerful features of the system.

At the moment everything seems quite sensible and we are all quite happy (at least I hope so). So this would seem a good point to confuse everybody and start to talk in more detail about XAML.

Why have XAML?

You might be wondering why we have XAML. If you have used Windows Forms before you might think that you managed perfectly well without this new language getting in the way. XAML provides a well-defined interface between the look of an application (the properties of the display elements that the user sees) and the behaviour of the application (what happens behind the display to make the application work).

The split is a good idea because, as we have mentioned before, there is no guarantee that a good programmer will be a good graphic designer. If you want to make good looking applications you really have to have both designers and programmers working on your solution. This leads to a problem, in that ideally you want to separate the work of the two, and you don't want the programmer unable to do anything until the graphical designer has finished creating all the menu screens.

XAML solves both these problems. As soon as the requirements of the user interface have been set out (how many text boxes, how many buttons, what is there on each page etc.) then the designer can work on how each should look, while the programmer can focus on what they do. The XAML files contain the description of the user interface components and the

designer can work on the appearance and position of these elements to their heart's content while the programmer gets one with making the code that goes behind them. There is no need for the programmer to give the designer their code files since they are separate from the design, and vice versa.

XAML file content

A XAML file for a given page on the display will contain constructions that describe all the elements on it. Each description will contain a set of named properties. The line of XAML that describes our first `TextBox` is as given below:

```
<TextBox x:Name="firstNumberTextBox" HorizontalAlignment="Left" Height="72"
Margin="10,0,-10,0" TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top"
Width="456"/>
```

If you compare the information in the Properties pane with the values in the XAML above you will find that they all line up exactly. The name of the `TextBox` is `firstNumberTextBox`, the width is `456` and so on.

If you were wondering how the page and the properties pane kept themselves synchronised it is because they both used the XAML description file for the page. When I move things around the page the editor updates the XAML with the new positions. The properties window reads this file and updates its values accordingly.

XAML is described as a *declarative* language. This means it just tells us about stuff. It is designed to be understandable by humans, which is why all the properties above have sensible names. If you want to, you can edit the text contents of the XAML files within Visual Studio and change both the appearance in the designer and also the property values.

XAML turns out to be very useful. Once you get the hang of the information used to describe components it turns out to be much quicker to add things to a page and move them about by just editing the text in the XAML file, rather than dragging things around or moving between property values. I find it particularly useful when I want a large number of similar elements on the screen. Visual Studio is aware of the syntax used to describe each type of element and will provide Intellisense support help as you go along.

When an application is built the XAML file is converted into a file containing low level instructions that create the actual display components when the program runs. This means that declaring a control inside the XAML file for a page will mean that the control will exist as an object our program can use.

XAML looks a lot like XML, an eXtensible MarkUp language you may have heard of. The way that items and elements are expressed is based on XML syntax and it is quite easy to understand.

Extensible Markup Languages

At this point you may be wondering what an eXtensible Markup Language actually is. Well, it is a markup language that is extensible. I'm sure that helped. What we mean by this is that you can use the rules of the language to create constructions that describe anything. English is a lot like this. We have letters and punctuation which are the symbols of English text. We also have rules (called grammar) that set out how to make up words and sentences and we have different kinds of words. We have nouns that describe things and verbs that describe actions. When something new comes along we invent a whole new set of word to describe them. Someone had to come up with the word "computer" when the computer was invented, along with phrases like "boot up", "crash" and "too slow".

XML based languages are extensible in that we can invent new words and phrases that fit within the rules of the language and use these new constructions to describe anything we like. They are called *markup* languages because they are often used to describe the arrangement of items on a page. The word markup was originally used in printing when you wanted to say things like "Print the name Rob Miles in very large font". The most famous markup language is probably HTML, HyperText Markup Language, which is used by the World Wide Web to describe the format of web pages.

Programmers frequently invent their own data storage formats using XML. As an example, a snippet of XML that describes a set of high scores might look as follows:

```
<?xml version="1.0" encoding="us-ascii" ?>
<HighScoreRecords count="2">
    <HighScore game="Breakout">
        <playername>Rob Miles</playername>
        <score>1500</score>
    </HighScore>
    <HighScore game="Space Invaders">
        <playername>Rob Miles</playername>
        <score>4500</score>
    </HighScore>
</HighScoreRecords>
```

This is a tiny XML file that describes some high score records for a video game system. The `HighScoreRecords` element contains two `HighScore` items, one for `Breakout` and one for `Space Invaders`. The two high score items are contained within the `HighScoreRecords` item. Each of the items has a property which gives the name of the game and also contains two further elements, the name of the player and the score that was achieved. This is quite easy to us to understand. From the text above it is not hard to work out the high score for the Space Invaders game.

The line at the very top of the file tells whatever wants to read the file the version of the XML standard it is based on and the encoding of the characters in the file itself. XAML takes the rules of an extensible markup language and uses them to create a language that describes components on a page of a display.

```
<TextBox x:Name="firstNumberTextBox" HorizontalAlignment="Left" Height="72"
Margin="10,0,-10,0" TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top"
Width="456"/>
```

If we now take a look at the description of a `TextBox` I think we can see that the designers of XAML have just created field names that match their requirements.

XML Schema

The XML standard also contains descriptions how to create a *schema* which describes a particular document format. For example, in the above the schema for the high score information would say that a `HighScore` must contain a `PlayerName` and a `Score` property. It might also say things like the `HighScore` can contain a `Date` value (the date when the high score was achieved) but that this is not required for every `HighScore` value.

This system of standard format and schema means that it is very easy for developers to create data formats for particular purposes. This is helped by the fact that there are a lot of design tools to help create documents and schemas. The .NET framework even provides a way by which a program can save an object as a formatted XML document. In fact the Visual Studio solution file is actually stored as an XML document.

As far as we are concerned, it is worth remembering that XML is useful for this kind of thing, but at the moment I want to keep the focus on XAML itself.

XAML and pages

A file of XAML can describe a complete page of the Windows Phone display. When you create a brand new Windows Phone project you get a page that just contains a few elements. As you put more onto the page the file grows as each description is added. Some elements work as *containers* which means that they can hold other components. They are very useful when you want to lay things out, for example there is a Grid element which can hold a set of other elements in a grid arrangement. The XAML code can also contain the descriptions of animations and transitions that can be applied to items on the page to make even more impressive user interfaces.

One thing you need to bear in mind at this point is that XAML has no concept of hierarchy. It is not possible to create an XAML element which is derived from another. We do this in software so that we can create a new object which is similar to an existing one. We know that elements are represented in software terms by an object which is part of a class hierarchy. However, when they are written in a XAML file they are all expressed as items which are at the same level.

We are not going to spend too much time on the layout aspects of XAML; suffice it to say that you can create incredibly impressive front ends for your programs using this tool. However, it is important to remember that at the end of the day the program is working in terms of objects that expose properties and methods that we can use to work on the data inside them. In the case of our user interface elements, if we change the properties of the element the display the user sees will change to reflect this. The objects represent the items on the screen of our program.

2.3 Putting Program Code into an Application

Now that we know that items on the screen are in fact the graphical realisation of software objects, the next thing we need to know is how to get control of these objects and make them do useful things for us in our application. To do this we will need to add some C# program code that will perform the calculation that the adding machine needs.

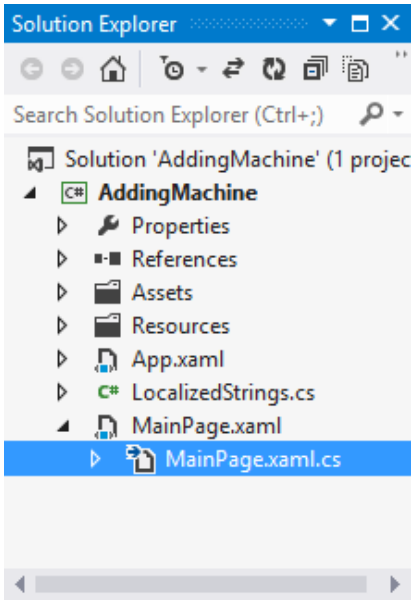


Figure 2-8 The Solution Explorer

Whenever Visual Studio makes a XAML file that describes a page on the Windows Phone display it also makes a program file to go with it. Figure 2-8 above shows how the C# source file for a page is located "underneath" the XAML file that describes that page.

This is where we can put code that will make our application work. If you actually take a look in the file `MainPage.xaml.cs` above you will find that it doesn't actually contain much code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Shell;
using AddingMachine.Resources;

namespace AddingMachine
{
    public partial class MainPage : PhoneApplicationPage
    {
        // Constructor
        public MainPage()
        {
            InitializeComponent();
        }
    }

}
```

Most of the file is `using` statements which allow our program to make direct use of classes without having to give the fully formed name of each of them. For example, rather than saying `System.Windows.Controls.Button` we can say `Button`, because the file contains the line `using System.Windows.Controls`.

The only method in the program is the constructor of the `MainPage` class. As we know, the constructor of a class is called when an instance of the class is created.

All the constructor does is call the method `InitializeComponent`. If you go take a look inside this method you will find the code that actually creates the instances of the display elements. This code is automatically created for you by Visual Studio, based on the XAML that describes your page. It is important that you leave this call as it is and don't change the content of the method itself as this will most likely break your program.

Note that at this point we are deep in the "engine room" of XAML. I'm only really telling you about this so that you can understand that actually there is no magic here. If the only C# programs you have seen so far start with a call of the `Main` method then it is important that you understand that there is nothing particularly special about a XAML model. There is a `Main` method at the base of a Windows Phone application; it starts the process of building the components and putting them on the screen for the user to interact with.

The nice thing as far as we are concerned is that we don't need to worry about how these objects are created and displayed, we can just use the high level tools or the easy to understand XAML to design and build our display.

Building the Application

We can now see how to create the user interface for our number adding program. If we add all the components and then start the application it even looks as though it might do something for us. We can type in the numbers that we want to add, and even press the equals button if we like:

Figure 2-9 The Adding Machine

Figure 2-9 shows the adding machine in action. If we touch inside a `TextBox` item the keyboard appears and we can type text and numbers into the box. If we then touch anywhere else on the screen the keyboard moves out of the way. We seem to have got a lot of behaviour for very little effort, which is nice. However, we need to add some business logic of our own now to get the program to work out the answer and display it.

Calculating the Result

At the moment our program looks good, but doesn't actually do anything. We need to create some code which will perform the required calculation and display the result. Something like this.

```
private void calculateResult()
{
    float v1 = float.Parse(firstNumberTextBox.Text);
    float v2 = float.Parse(secondNumberTextBox.Text);

    float result = v1 + v2;

    resultTextBlock.Text = result.ToString();
}
```

The `TextBox` objects expose a property called `Text`. This can be read from or written to. Setting a value into the `Text` property will change the text in the textbox. Reading the `Text` property allows our program to read what has been typed into the textbox.

The text is given as a string, which must be converted into a numeric value if our program is to do any sums. You may have seen the `Parse` method before. This takes a string and returns the number that the string describes. Each of the numeric types (`int`, `float`, `double` etc.) has a `Parse` behaviour which will take a string and return the numeric value that it describes. The adding machine that we are creating can work with floating point numbers, so the method parses the text in each of the input textboxes and then calculates a result by adding them together.

Finally the method takes the number that was calculated, converts it into a string and then sets the text of the `resultTextBlock` to this string. `ToString` is the reverse of `Parse`, the "anti-parse" if you like. It provides the text that describes the contents of an object. In the case of the float type, this is the text that describes that value.

Now we have our code that works out the answer, we just have to find a way of getting it to run when the user presses the equals button.

Events and Programs

If you have done any kind of form based programming you will know all about events. If you haven't then don't worry, we now have a nice example of a situation where we need them, and they are not that frightening anyway. In the olden days, before graphical user interfaces and mice, a program would generally start at the beginning, run for a while and then finish.

But now we have complicated and rich user interfaces with lots of buttons and other elements for the user to interact with. The word processor I am using at the moment exposes hundreds of different features via buttons on the screen and menu items. In this situation it would be very hard to write a program which checked every user element in turn to see if the user has attempted to use that. Instead the system waits for these display elements to raise an event when they want attention. Teachers do this all the time. They don't go round the class asking each child in turn if they know the answer. Instead they ask the children to raise their hands. The "hand raising" is treated as an event which the teacher will respond to.

Using events like this makes software design a lot easier. Our program does not have to check every item on the screen to see if the user has done anything with it, instead it just binds to the events that it is interested in.

To make events work a programing language needs a way of expressing a reference to a method in an object. C# provides the *delegate* type which does exactly that. You can create a delegate type that can refer to a particular kind of method and then create instances of that delegate which refer to a method in an object. Delegates are very powerful, but can be a bit tricky to get your head around. Fortunately we don't have to worry about how delegates work just at the moment; because we can get XAML and Visual Studio to do all the hard work for us.

Events in XAML

In C# an event is delivered to an object by means of a call to a method in that object. In this respect you can regard an event and a message as the same thing. From the point of view of our adding machine we would like to have a particular method called when the "equals" button is pressed by the user. This is the only event that we are interested in. When the event fires we want it to call the `calculateResult` method above.

We can use the editor in Visual Studio to bind this event for us. This is actually so easy as to appear magical. To connect a method to the click event of a particular button we just have to double click on that button on the design surface.

Figure 2-10 Binding an event

The figure above shows where we should double click in Visual Studio. When we double click on the button Visual Studio sets the properties of the button in XAML to connect the button to a method in our page. It also creates an empty version of that method and takes us straight to this method, so that we can start adding the code that must run when the button is clicked.

If we just single click on the button this has the effect of selecting it, so that we can move it around the display and change its size. If you want to connect an event handler it must be a double click. If we do this we will see a display like the one shown below.

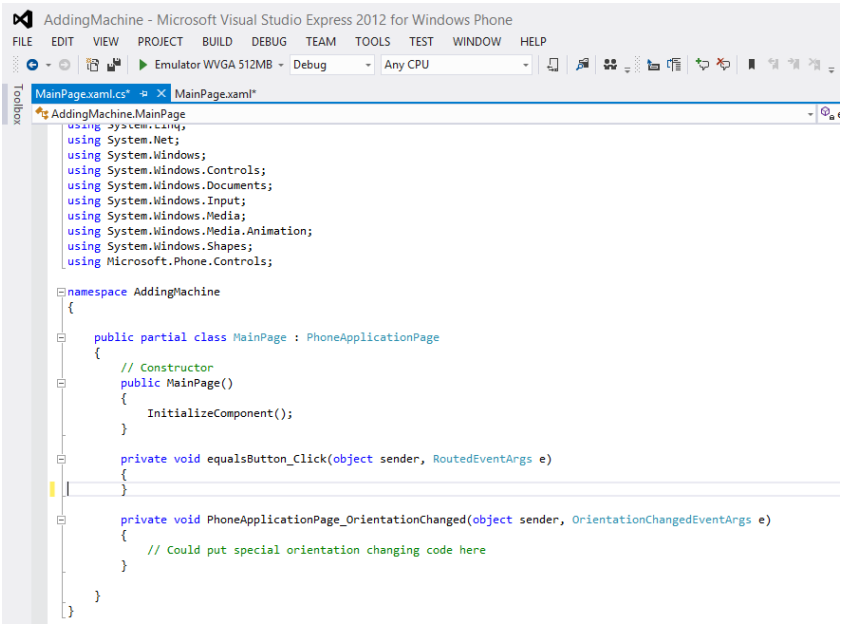


Figure 2-11Method bound to the event

Figure 2-11 shows the result of "double clicking" the button on the design surface. Now, all we have to do is add the code to make this work.

```
private void calculateResult()
{
    float v1 = float.Parse(firstNumberTextBox.Text);
    float v2 = float.Parse(secondNumberTextBox.Text);

    float result = v1 + v2;

    resultTextBlock.Text = result.ToString();
}

private void equalsButton_Click(object sender, RoutedEventArgs e)
{
    calculateResult();
}
```

The event handler that is created is automatically given a sensible name by Visual Studio. It takes the name of the element and adds the text "_Click" on the end. This underlines the importance of giving your elements sensible names when you create them, because this name makes it very easy for anyone to understand what is happening.

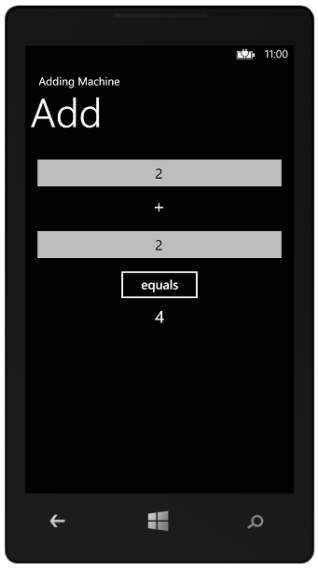


Figure 2-12 Adding machine in use

The program works fine, as you can see above. Of course it is not perfect. If a user types in text rather than a numeric value the program will fail with an exception, but we will deal with this later.

Managing Event Properties

At the moment the process of connecting event handlers to events looks a bit magic. And if there is one thing I'm sure of, it is that computers are not magical. Ever. So now we should really find out how this event process actually works. We can start by taking a look at the properties of the equals button in our application.

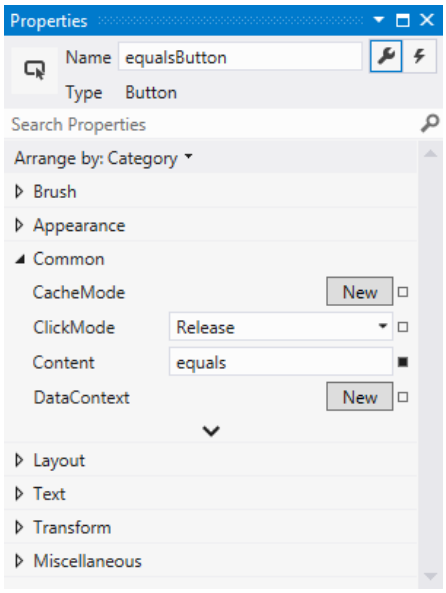


Figure 2-13 Button Properties

If we single click on the button in the editor and then take a look at the property information in Visual Studio we see something very like Figure 2-13 above. The information here sets out the position of the button on the screen and a bunch of other things too. The property window has two tab panels though. At present we are looking at the "standard" properties of the events, as shown by the highlighted spanner in the top right hand corner.

If we look at the top we also find a tab with a rather scary looking lightning bolt icon. If we click on this tab the display changes to display the events that this element can generate, and which are presently connected to code:

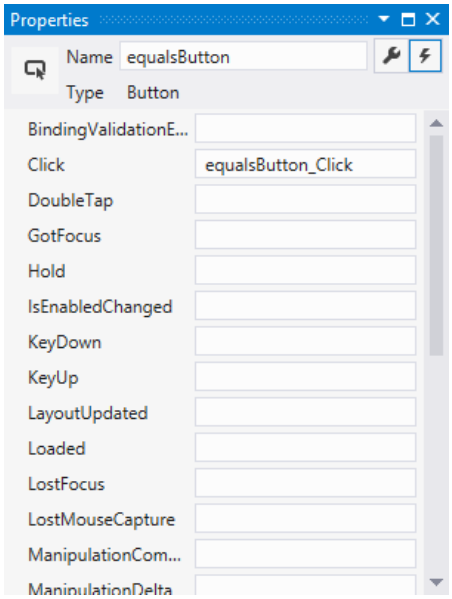


Figure 2-14 Showing connected events

The Properties window is now showing us all the events that a button can generate. At the moment only one event has a method connected to it, and that is the Click event. This is linked to the `equalsButton_Click` method. If we wanted to disconnect the Click event from the method we could do this by just deleting the name of the handler from this window. We could cause problems for ourselves by replacing the content of that part of the form with something stupid like "ILikeCheese". If we do this the program will not work (or even build) anymore because the Click event is now connected to something that does not exist.

Events and XAML

Up until now we have worked on the principle that the properties display in Visual Studio is driven by the XAML file which describes the components on the screen. We can prove that this is the case by taking a look at the Button description in the XAML file for the MainPage:

```
<Button Content="equals" Height="72" HorizontalAlignment="Center"
Name="equalsButton" VerticalAlignment="Top" Width="160" Click="equalsButton_Click" />
```

As you can see; the description of Button now contains an item that connects the Click event with the name of a method.

One important point to note here is that if the XAML file says that a button is connected to a method with a particular name the program will fail to run correctly if that method is not there.

What We Have Learned

1. XAML (eXtensible Application Markup Language) provides a way of designing graphical user interfaces.
2. A Windows Phone application user interface is made up of elements which represent things such as text boxes and buttons.
3. The Visual Studio integrated development environment provides an editor which can be used to add XAML user interface elements onto the pages of an application. There is also another editor, called Blend, which can be used to perform more advanced graphical design tasks.
4. From a software point of view each of the user interface elements is represented by a particular type of object in a class hierarchy which is part of XAML.
5. Designers can modify the properties of the elements using the design tools in Visual Studio, either by changing them on the design surface or by modifying their properties.
6. The actual properties of XAML elements in a project are held in text files in the XAML format. These files are updated by the design tools and used when the program is built to create the software objects that are used in the solution.

7. XAML (eXtensible Application Markup Language) is an XML based language that specifies all the properties of the design elements on a page. It provides a separation between the appearance and properties of the display elements and the program code that sits behind them.
8. XML (eXtensible Markup Language) is a way of creating customised languages that can be used to describe things.
9. Elements can generate events which may be bound to methods inside a C# program. The method name is given in the XML description for the element.
10. The methods that are bound to the events can provide the business logic for a user application.

3 Visual Studio Solution Management

When you write programs for Windows Phone you will be using Visual Studio. In this section we will take a look at the process of creating and managing Windows Phone projects. We will also find out how to run and debug Windows Phone programs using the Windows Phone emulator program. This allows you to test your programs without needing to have a device.

However, this is not just a look at how to use Visual Studio. That would be too easy. We are also going to take a peek underneath the hood and find out how Visual Studio manages the content that makes up our solutions. This will stand us in good stead when we come to create Windows Phone applications.

3.1 Getting Started with Projects and Solutions

The first thing we are going to look at is just how Visual Studio manages the programs that you create.

Creating programs

We know that when we create a program we write a set of instructions in a high level language (which if we are lucky is C#). If you would rather work with Visual Basic .NET than C# you will be pleased to discover that you can create Windows Phone programs in that language too. This text is written for C# developers, but all the fundamental principles and the application programmer interface (API) calls can be mapped directly into the Visual Basic Language.

The *compiler* is a program that takes the high level instructions which the programmer created and converts them into lower level instructions for execution on the computer. We also know now that in Microsoft .NET the compiler produces output in an *intermediate language* (MSIL) which is then "Just In Time" compiled into hardware instructions when the program actually runs.

The bare minimum you need to create a working program is therefore a C# source file (a text file with the language extension .CS) and a compiler. The compiler takes the source file and converts it into an executable file (a binary file with the language extension .EXE) which can then be used by the .NET runtime system for whatever platform you want to execute the program on. The use of MSIL means that I can take exactly the same .EXE file and run it on completely different hardware platforms if I wish.

The .NET SDK

If you just want to create programs for the Windows PC and you don't want to download all of Visual Studio you can actually just get the compiler and runtimes for .NET from here:

http://msdn.microsoft.com/en-us/netframework/aa569263.aspx

When you install the framework you get a command line compiler which can take C# source files and convert them into executable ones using console commands such as:

```
csc MyProg.cs
```

This command will compile the file `MyProg.cs` and create a file called `MyProg.exe`.

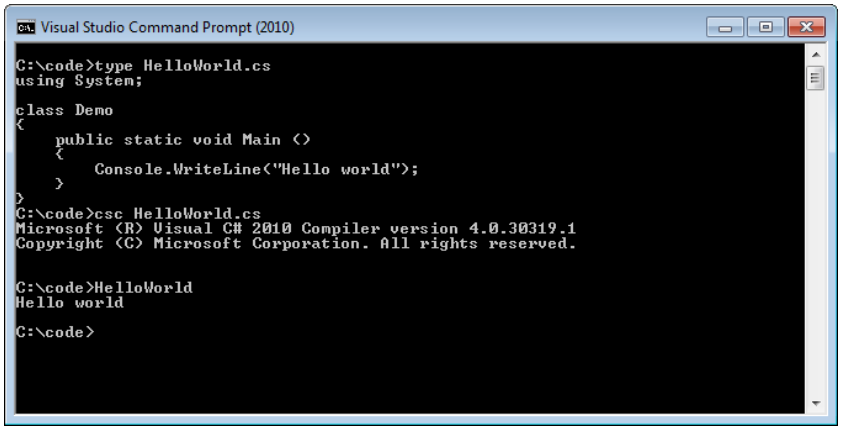


Figure 3-1 Compiling and running programs at the command line

Figure 3-1 shows how a program can be compiled and executed from the command line. The program source has been created using Notepad and typed into a file called HelloWorld.cs. This is then compiled to produce an output file, HelloWorld.exe, which contains a runnable version of the program. If we then give the name of the file as a command the program is loaded and executed for us.

It is possible to use these simple tools to create very large programs, but it is also very hard work. You have to create large, complicated commands for the compiler and you need to make sure that whenever you make a new version of your system that you compile every source file that you need to. Furthermore you have to manage and incorporate all the media that your application needs and when it comes to debugging your program all you are going to get is text messages when the program fails.

The Windows Phone SDK

You don't need to download the .NET SDK if you want to create Windows Phone applications using Visual Studio. For that you just need to go to:

```
create.msdn.com
```

This site contains links to the latest versions of the Windows Phone SDK and you can also register here to sell your programs in the Windows Phone Store. Note that if you are a student you can get free access to the store via the DreamSpark program. Dreamspark can also give you access to free copies of the Professional versions of Visual Studio.

```
www.dreamspark.com
```

The Windows Phone SDK is all installed at once. Once the installer has finished you will have the Interactive Development Environment, the Windows Phone emulator and other tools to help you create Windows Phone applications. The tools can also be used to create desktop applications and XNA games for Windows PC and Xbox 360. If you have another version of Visual Studio 2010 present when you install the Windows Phone SDK it will add the new project types and the tools alongside that version.

Visual Studio

Visual Studio provides a single environment in which you can edit, build and run your programs. It manages projects and solutions which let you organise your systems and control what goes into them. Finally, just to make it even more wonderful, it provides a full debugging solution that lets you single step through your program and even execute individual statements and change the values in variables as your program runs. Skill with Visual Studio and an understanding of how it works is very marketable. Remember that this tool is used to create PC applications as well as those for the Windows Phone.

Once you have installed the Windows Phone SDK you will find Microsoft Visual Studio Express 2012 in your programs menu. Simply start that to begin writing programs.

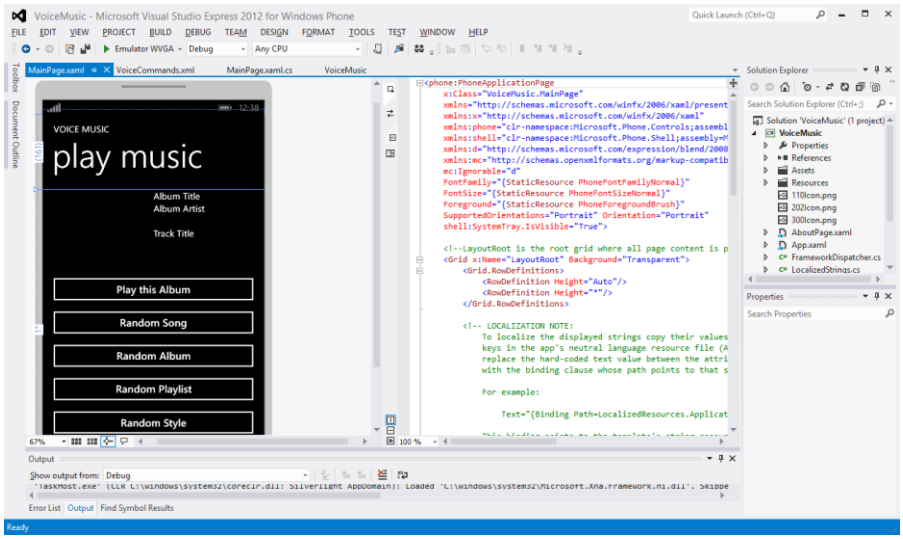


Figure 3-2 Visual Studio Express 2012 display

The first thing that hits you with Visual Studio is the sheer number of buttons and controls that you have to deal with. If we start opening menus it gets even worse. However, the good news is that you don't have to press all the buttons to get started. We are just going to take a look at the basics of the program. You can pick up more advanced features as we go along.

The first aspect of Visual Studio we are going to take a look at is that of projects and solutions. You will use these to organise any systems that you create. Visual Studio is not able to run any program code which is not in a project or solution. The good news is that to get started you can use one of the templates that provide a starting point for particular project types. We will be using the Windows Phone templates for now. There are many other templates for different kinds of programs.

Visual Studio makes a distinction between a *project* and a *solution* when organising your work. A solution is the outermost container that holds one or more projects.

The adding machine that we saw in the previous chapter was created as a single page Windows Phone application. If you take a look at Solution for this application you see this:

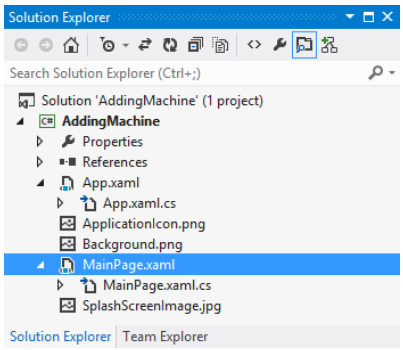


Figure 3-3 Solution Explorer for a Windows Phone application

The solution (which is called AddingMachine) contains a single project (which is, rather confusingly, also called AddingMachine). Inside the AddingMachine project are all the elements that make up the Windows Phone application, including the various splash and background screens.

A Windows Phone project contains the `MainPage.xaml` file that describes the appearance of the main page (hence the name). The main page is simply the first page that the application displays when it starts. A project can contain further pages, each of which will have its own XAML file. Later in this text we will discover how to *navigate* from one page to another.

Creating a Windows Phone solution

We have seen how to create a very simple application which just contains a Main method. The process of creating a Windows Phone application is exactly the same. We choose a template and then Visual Studio creates the projects and source files that match that template.

The starting point is `File>New>Project` from within Visual Studio Express 2012. This produces the New Project dialogue as shown in Figure 3-4Figure 3-4:

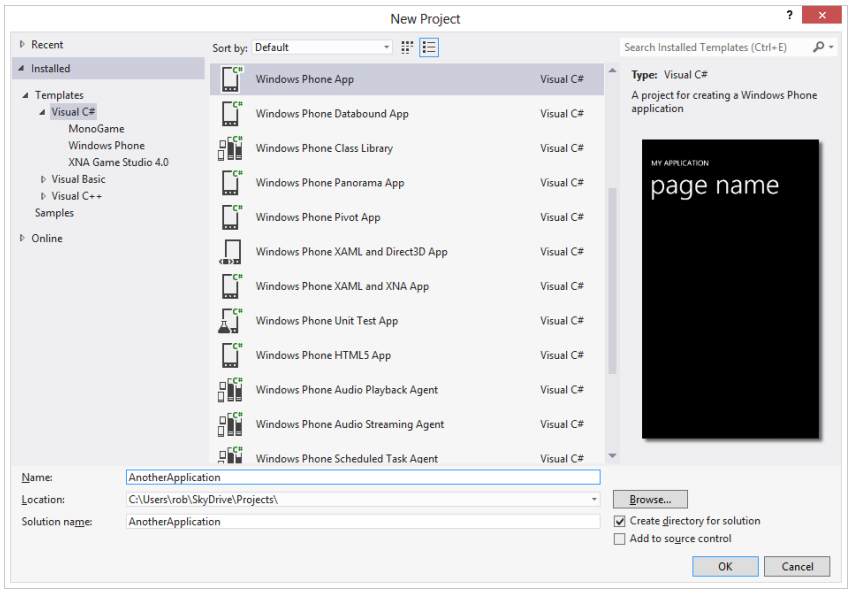


Figure 3-4 Creating a new Windows Phone Project

On the left of the dialog box we can see all the project types available. This screenshot was taken from my "working" machine, as you can see I've installed some extra toys, including MonoGame. You might not have those present, but you should have "Windows Phone App" available, along with the ones for "XNA Game Studio 4.0". If you haven't this means you are using a machine that does not have the Windows Phone SDK installed on it. The SDK is a free download from `dev.windowsphone.com`.

Above in the middle window you can also see the range of Windows Phone projects available. On the right hand side is a preview of what such an application would look like.

We can also use a template to create a project that is added to an existing Windows Phone application. We do this when we have a special task to perform. For example we might add the "Windows Phone Scheduled Task Agent" project to a solution. You would never create a new solution that just contains this project, but you might add a scheduled task to your application so that it can perform actions at regular intervals when the user is not actually using your application.

To start with we are going to create a simple "Windows Phone Application" using the top template. There are other templates you might want to explore later. Note that there is nothing particularly special about a template; it just creates certain combination of projects and source files. You could actually create the same effect by adding projects and source files and adjusting the settings, but the templates make it much easier.

We need to type in a name for the application. In the screenshot above I've used the rather daft name `AnotherApplication`. Note that if you have the "Create Directory for Solution" box checked (and you should) Visual Studio will create a folder with the application name and place all the solution files and project folders into it.

When we click OK Visual Studio asks us what kind of application we want to create

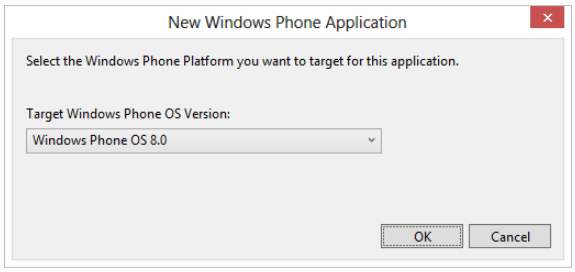


Figure 3-5 Selecting the target for an application

The first version of Windows Phone was called, rather confusingly, Windows Phone 7. There is nothing new in manufacturers playing around with version numbers. This has been going on ever since Version 3.0 of Fire was introduced way back in cave man days. The latest version of Windows Phone is Windows Phone 8. You can also use this tool chain to create an application for Windows Phone 7.5 (rather confusingly using Windows the Windows Phone 7.1 toolchain).

The only reason for creating a Windows Phone 7.5 application is so that you can target owners of older phones as well as Windows Phone 8 owners. However, this text is going to focus on Windows Phone 8 development as there are lots of useful extras you get in the latest version of the platform.

After you click OK to select your Windows Phone OS version Visual Studio will set to work building an empty Windows Phone application. The application will run, and you can even load it into a Windows Phone, but it won't be very interesting. It is simply a placeholder, into which you can add the display elements and behaviours that make the application into a useful thing to have.

Running Windows Phone applications

Starting the Compile and Run process in Visual Studio is as simple as pressing the `F5` function key, or selecting `Debug>Start Debugging` from the menu. When this happens Visual Studio will assemble all the resources required to make the program work and begin running it. We will find out more about this process when we debug a program in the next section.

We know that when you compile and build a Windows PC application the result will be a file with a language extension of `.exe` which contains a method named `Main` which will be called when the program starts running.

If you compile and build a Windows Phone application things are a little more complicated. The program will not be running inside the PC, instead it must be transferred into the Windows Phone device, or the emulator, before being started. Furthermore, any resources that the program needs (for example content that is not part of the program assembly) must be made available in the target device. This problem is solved by the use of a container file that holds the entire application and all required resources. Below you can see the output directory for the AddingMachine program. It contains all the resources along with the AddingMachine.dll file that contains the program code.

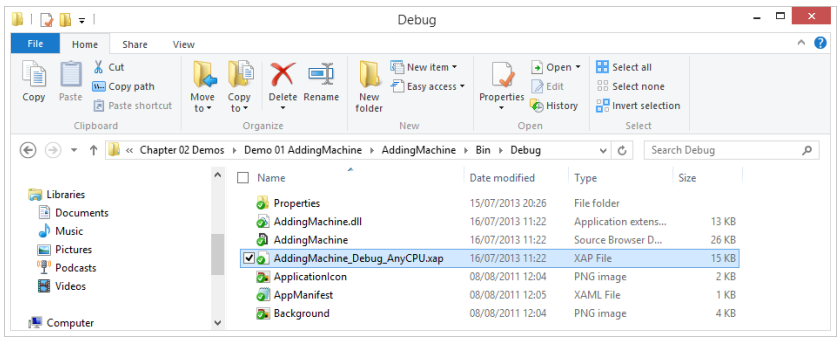


Figure 3-6 Compiler output from a build

There is also a file that has the language extension `.XAP`. This is the *XAP* file. This is the container that holds the entire application.

The XAP file

The XAP file is an archive file (stored in the same format as Zip files) which contains all the files that make up the application. It also contains a manifest file which describes the contents of the XAP.

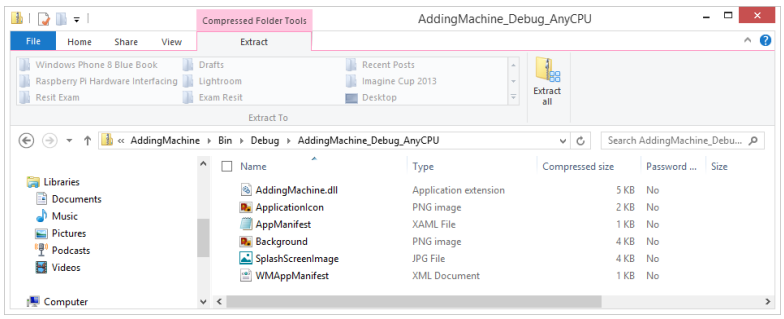


Figure 3-7 XAP file contents

Figure 3-7 shows the contents of the XAP file for the `AddingMachine` application. The XAP file is the file that is transferred into the Windows Phone device and made to run. When you press the run button in Visual Studio this file is created and then sent into the device or emulator. The target device opens the file, reads the manifest and then unpacks the content. Finally the application is executed. If your program is made up of multiple libraries and contains images and sound files these will all placed in the XAP file to be picked up in the target.

When you submit an application to the Windows Phone Store you actually send the XAP file, this is the file that is loaded into the customer's phone if they buy your program.

3.2 Debugging Programs

One of the great things about the Windows Phone development environment is the ease with which you can debug programs. We can get a really good idea of what is going on inside a program by single stepping through the code and taking a look at the values in variables. You can debug programs on either the real device or the emulator.

To start a program we press `F5` function key or select `Debug>Start Debugging` from the Visual Studio menus. We could also press the green triangular start button that you can see on the screenshot below. But before we do that, we want to decide whether we are running the program on a real device or the emulator.

Using the Windows Phone emulator

The Windows Phone emulator gives you a chance to try your programs and discover what they look like when they run. The emulator works in just the same way as a real phone. Effectively it is a Windows PC implementation of the Windows Phone platform, running exactly the same source code as the actual phone. You can use the emulator to prove that your program works correctly.

Deploying to the emulator

You can deploy programs to the emulator or a real device. To select the destination when you run a program you use the combo-box to the right of the run button as shown below.

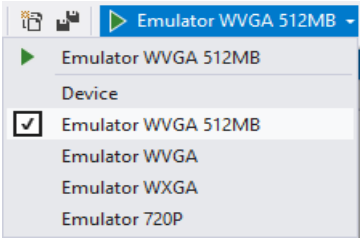


Figure 3-8 Windows Phone Emulator options

There are a number of different emulators available for selection. Each of them has a particular screen resolution, so you can test that your program will work fine on all the different devices available. There is also a version of the emulator that is "fitted" with only 512MB of memory. You can use this to make sure that your program works on lower specification devices.

It is possible to select a Windows Phone device even when one is not connected. In this case you will get an error when you try to run the program and Visual Studio detects that a device is not attached.

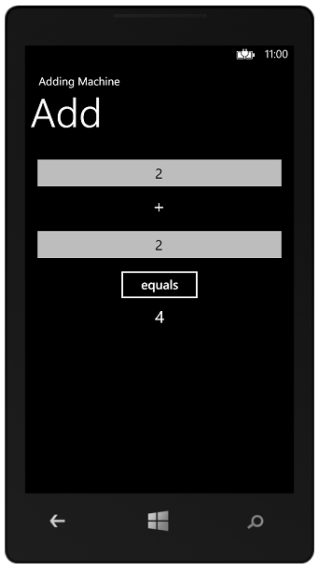


Figure 3-9 Running the Adding Machine in the emulator

When the program starts it runs in exactly the same was as it would on the Windows Phone device. You can use the Windows PC mouse to click on items. If you have a multi-touch monitor on your PC you can use multiple touches on the emulator screen to access the multi-touch features of the user interface.

To stop the program you can press the red stop button in Visual Studio.

Emulator features

Note that the emulator is provided with all the built in programs provided with a real device. You can browse the internet using Internet Explorer and you can change device settings. Each program that you run on the emulator will stay in the program memory of that emulator until you stop the emulator program.

Emulator performance

The emulator cannot run at exactly the same speed as the Windows Phone hardware. The actual display speed of an application is restricted on the emulator so that the display will match the real device to a certain extent, but it is important to run your programs on a real device to get an accurate impression of what the user experience will be.

Programs in the emulator

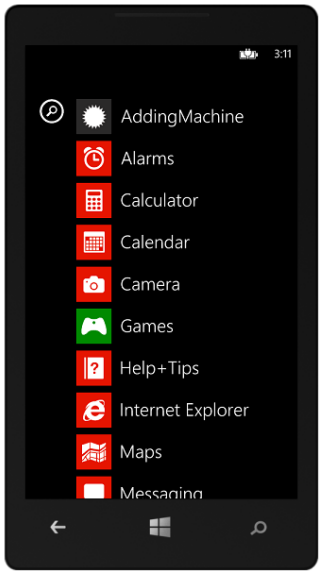


Figure 3-10 Programs stored in the Windows Phone emulator

When Visual Studio deploys an application to the Windows Phone emulator it installs it on the emulated device. The application will remain there until the emulator is reset. Above you can see the start menu for the emulator after the `AddingMachine` application had been run on it. We can run this application again just by selecting it. Note that if you do this you will not be able to debug the application, you will just be running it.

Visual Studio Debugging

The debugging features are integrated into the Visual Studio environment. You can start programs running from within Visual Studio and add *breakpoints* into your program. When a program reaches a statement nominated as a breakpoint Visual Studio will pause the program and allow you to look at the contents of variables inside the code.

You can add breakpoints to programs running inside the Windows Phone device. You can even set and remove breakpoints as the program itself is running.

Debugging is an important part of the development process. If you are going to become a successful programmer you are going to have to get used to that feeling in the pit of your stomach that you get when the program doesn't do what you expected. Fortunately there are some even more powerful tools and techniques that you can use to debug your programs.

Debugging and Programming

However, one important thing to remember here is that I don't consider it acceptable to debug your programs into life. Every line of code you write should be put there on the basis that you know exactly what the line is going to do, and why it is there. I use debugging to fix an implementation of a solution I have fully thought through. My programs tend to go wrong because I have miss-typed something or because I don't have the correct understanding of the behaviour of a library I'm using. If you don't know how to solve the problem it is very unlikely that throwing a few lines together might just solve the problem, any more than throwing a bucket of components at a wall will get you a new Xbox.

Adding a breakpoint at a statement

Breakpoints are added in just the same way as for any program you are working with in Visual Studio. You click in the left margin of the line you want the program to break at.

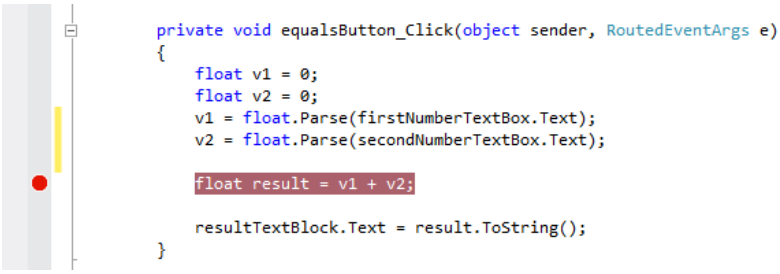
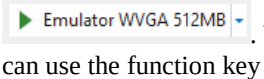


Figure 3-11 Creating a breakpoint

The line is highlighted as shown in Figure 3-11 above. You can clear a breakpoint by repeating the process. When the program reaches the breakpoint it will pause and you will be able to take a look at the values of the variables. You can then resume the program.

Running the Program

We run the program by clicking the Run button in the Visual Studio menus. The button looks like this: 

. When you press the button the program is deployed into the target device and starts running. You can use the function key F5 to instead of pressing the button if you prefer. When the program runs, if it reaches this line the execution will pause. Of course, to get to this statement in the adding machine we will have to enter some numbers into the textboxes and then press the Calculate button.

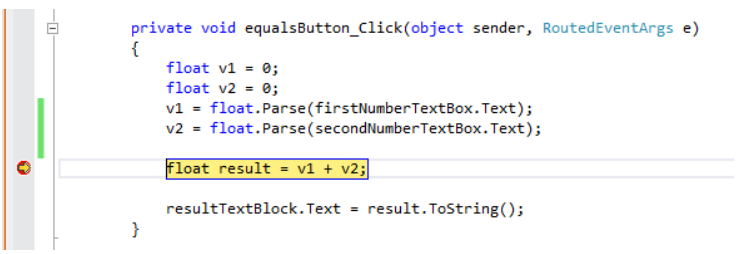


Figure 3-12 Hitting a breakpoint

When the program hits a breakpoint the yellow highlight shows the line reached. We can view the contents the variables in the program at this point just by resting the mouse pointer on the variable we want to see:

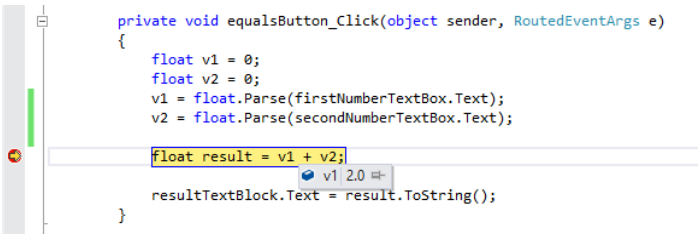


Figure 3-13 Viewing the contents of a variable during debugging

The variable `result` has not been set to the result of the calculation yet because the line that is highlighted has not been performed. To get the line performed we need to step through the statement.

Single stepping

We can step through a single line of the program by clicking the Single Step button in the Visual Studio menu system. The button looks like this: .

Each time you click this button the program will obey a single statement. Clicking the button once will move the program onto the next statement. The function key F11 will have the same effect.

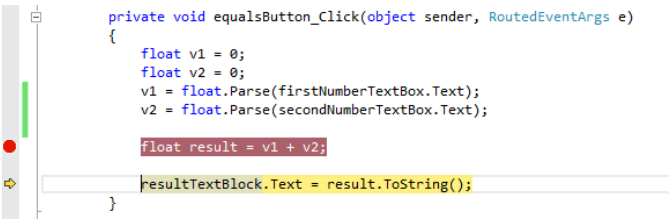


Figure 3-14 Running on after a breakpoint

There are actually three versions of the Single Step button.

This means obey a single statement. If the single statement is a call of a method this version of single step will step into the method. This is called the "Step into" version of single step. Use F11 to perform this.

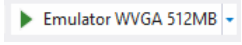
This means obey a single statement. If the single statement is a call of a method this version of single step will step over the method call. This is useful if you don't care what happens inside the method. This is called the "Step over" version of single step. Use F10 to perform this.

This means run to the end of the current method and exit from it. Then break. This is very useful if you step into a method by mistake, or you have seen all you need from the method you are stepping through. Use SHIFT+F11 to perform this.

Controlling Program Execution

Visual Studio maintains contact with the running program in the target device. You can issue commands to stop, pause and resume execution and these will directly affect the program.

Starting or Resuming Execution

If you want to start the program you press the 

(run) button. You also use this button to resume execution after a breakpoint. You can also press the F5 function key.

Pausing program execution

Pausing execution is not usually that useful, in that the program might not be at a place in the code where it is doing anything interesting, but you can sometimes use it to find out what a program is doing when it appears to have got "stuck".

To pause a running program you click the  button. The program stops at the statement that it was running when you pressed pause.

If you want to pause a running program you should remember that you can set breakpoints in the program code even when the program itself is running. We could have put our breakpoint in the `calculateResult` method while the code was running and then pressed the equals button in the emulator to send the program that way.

Stopping the program

The  button can be used to stop the program from running. When we click this button Visual Studio will stop program after the next statement it obeys. We can use it if the program appears to have got stuck. We need to be a bit careful with this button though.

When a Windows Phone program is stopped properly it is sent a sequence of messages that tell it the end is nigh and ask the application to do any tidying up that is required. We will discover how this works later. If the program is stopped using the button above these messages are not sent, and so a program interrupted in this way may lose or corrupt data that it was using when the stop event was fired.

Managing Breakpoints

Visual Studio also has a breakpoint window that you can use to see the breakpoints you have created. You can display this window by going to the `Debug>Windows` menu in Visual Studio and selecting the `Breakpoints` option.

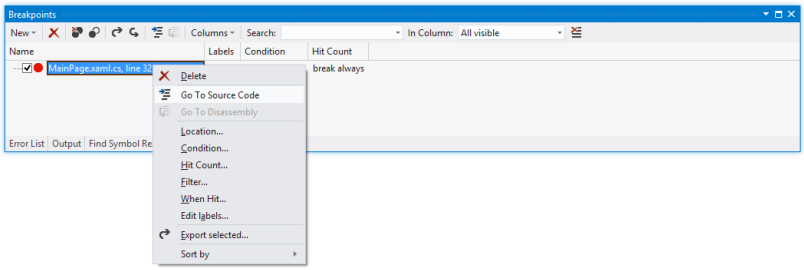


Figure 3-15 Managing a Breakpoint

Figure 3-15 shows the breakpoint that we created earlier, displayed in the Breakpoints window. You can use this window to set breakpoint properties. You can request that a breakpoint only fire after a given number of hits, or only fire when a particular condition fires. This is terribly useful. You might have a program that fails after the fiftieth time round a loop. It is very nice to be able to add a counter or a test so that the program skips past the breakpoints you are not interested in.

To open the properties of a breakpoint, either in the code itself or in the Breakpoints window, right click the breakpoint and then select Breakpoint from the context menu that appears.

Using the immediate window

The Immediate Window can be displayed by going to the `Debug>Windows` menu and selecting it. It is where you can view and change the contents of variables. You can also use it to evaluate expressions. As an example of the immediate window in action, we could set a breakpoint just before the `calculateResult` method returns:

```
private void equalsButton_Click(object sender, RoutedEventArgs e)
{
    float v1 = 0;
    float v2 = 0;
    v1 = float.Parse(firstNumberTextBox.Text);
    v2 = float.Parse(secondNumberTextBox.Text);

    float result = v1 + v2;

    resultTextBlock.Text = result.ToString();
}
```

Figure 3-16 Hitting a breakpoint

When I run the program and press the equals button the program will hit the breakpoint as shown in Figure 3-16 . I can now move to the Immediate Window and play with the variables in this method.

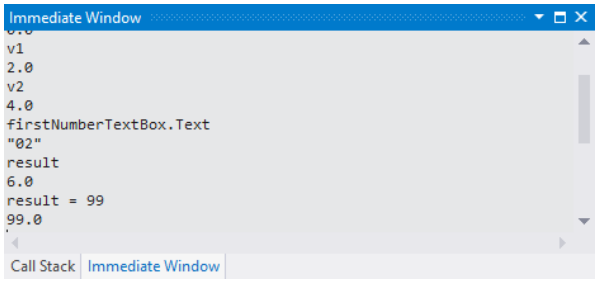


Figure 3-17 Working with the variables in the program

If I just type the name of a variable the window displays the value in that variable. I can also type in expressions and these are evaluated and displayed. I can also assign a value (e.g. `result = 99`) and call methods on objects to work out results.

The Immediate window uses Intellisense to help you write the statements you want to run. It even works when you are debugging a program stored inside the Windows Phone device. It is very powerful, but should be used with care.

Design for debug

Single stepping through code is a great way to find out what it is doing. When you write code it useful to design it so that you can easily step through it and find out what it is doing. The code in the `calculateResult` method could easily be as a single statement.

```
private void calculateResult()
{
    resultTextBlock.Text =
       (float.Parse(firstNumberTextBox.Text) +
        float.Parse(secondNumberTextBox.Text)).ToString();
}
```

However this version is impossible to step through. I am a great believer in spreading my code over a number of statements and even using extra temporary variables when it runs. This makes debugging easier and doesn't even cost your program more memory since in the "smaller" version of the code above the compiler will still have to create some internal temporary variables to perform the calculation. The only difference with the "more efficient" version above is that you can't take a look at them.

3.3 Performance Tuning

Performance tuning is something you can do during development to make sure that your program gives the user a good experience and doesn't flatten the battery too quickly. The Windows Phone SDK includes a profiling tool that you can use to find out what demands your program is placing on the phone, and where the program is spending most of its time. You can also find out how much memory the program is using, and how often the Garbage Collector is being used to remove discarded objects from memory. You can use the performance analysis tool in Visual Studio to find out the demands that your application places on a device and spot situations where it may consume resources when it shouldn't, for example memory leaks.

What We Have Learned

1. Visual Studio is the development environment that is used to create Windows Phone applications. You can obtain a free version of this SDK.
2. A Visual Studio solution brings together a number of projects which can describe the creation of different elements of an application. This can include the production of completely different components, for example the server and client parts of a large installation.
3. A Windows Phone application can contain a number of assemblies, including library resources. An assembly will contain versioned references to all the resources that it uses.
4. When a Windows Phone application is prepared for deployment to the target device it is packaged into a XAP file which contains a manifest file, program assemblies and any resources that the program uses.
5. The Windows Phone emulator runs on the Windows PC and provides an emulation of the functionality of the device, but does not emulate the performance.

6. Visual Studio provides a means by which a programmer can insert breakpoints into program. These are statements where program execution will stop and the programmer can view the contents of the variables in the program. Breakpoints and single stepping of code can be used on programs executing in the Windows Phone device as well as the emulator.
7. The Windows Phone Performance tools allow a developer to find and target those parts of their program that could best be optimised for speed. It also helps to detect “memory leaks” and allows a developer to see how many “garbage” objects it is creating as it runs.

4 Constructing a program with XAML

In chapter 2 we looked at some of the elements provided as part of the XMAL framework. Now we are going to build on our skills and delve a bit more deeply into what we can do with these elements. By the end of this section you should be able to create useable, multi-page applications that will work on Windows Phone and allow the user to manipulate data. This is a large chapter, with a lot of demonstrations. But don't worry.

4.1 Improving the User Experience

At the moment we are able to build user interfaces out of three elements:

- TextBlock to display messages and labels
- TextBox to receive input from the user
- Button to receive events

Now we are going to build on these three items to create slightly more interesting applications. At the same time we are going to learn a bit more about how the XAML elements work together to build user interfaces.

Manipulating Element Properties

We have seen that XAML display elements are actually implemented as classes in the code, and that we can change how they appear by modifying the properties in instances of these elements. The properties of an element can be set using the Properties pane in Visual Studio or by editing the XAML that describes each element. However, these settings are made when the program is built. We can also set the properties of the elements when the program runs. We have already done this in our `AddingMachine` application, the result is displayed by changing the properties of the `ResultTextBlock`.

```
resultTextBlock.Text = result.ToString();
```

Programs can manipulate any of the properties of a display element. This includes their position on screen, making it possible for us to create moving sprites in games.

We are going to take a look at some other properties that are available. We are going to use further property manipulation to improve the user experience of our `AddingMachine` program. However, we are just going to scratch the surface of what is actually possible. I strongly advise to you spend some time looking at just what you can do with elements. There are some very powerful features available.

Adding Error Handling

We are going to start by improving the error handing of our `AddingMachine`. In my experience I spend at least as much time writing code to deal with invalid inputs for a system as I do writing programs that deal with the working parts.

At the moment the error handling in our `AddingMachine` program is non-existent. If a user enters text into a `TextBox` the program will try to use the `Parse` method to convert this into a number, and this will fail.

Figure 4-1 Entering text versions of numbers

In Figure 4-1 you can see trouble brewing for our adding machine program. The user will claim that they just wanted to work out the answer to “two plus two” but the result will not be a successful one:

Figure 4-2 Number format error

Figure 4-2 shows the error that is produced by Visual Studio when you press Calculate with text in the input fields. If the user was just running your program on their phone they would not see this, they would just notice that your program had not calculated the result they were expecting.

The problem is caused because this version of the program uses a very simple method to convert the text in the `TextBox` into a number:

```
float v1 = float.Parse(firstNumberTextBox.Text);
```

If `firstNumberTextBox` contains text that cannot be converted into a number (for example “two”) it will throw an exception. Your program could catch the exception and display an error message or can use a different version of the `Parse` method which returns an error rather than throwing an exception:

```
float v1 = 0;
if (!int.TryParse(firstNumberTextBox.Text, out v1))
{
    // Invalid text in textbox
}
```

The code above uses the `TryParse` method which returns `false` if the parse fails. It will obey the statements in the conditional clause if the text in `firstNumberTextBox` contains invalid text. It would be nice to turn the text red in the `TextBox` to indicate there is a problem.

Changing the colour of text

A good way to show an error status is to change the colour of the text. This is a very good way to highlight a problem. The application uses a system of brushes to draw items on the screen. This is a very powerful mechanism. It means that we can draw text using images or textures and get some very impressive effects. In the case of our error display we don't want to do anything quite so interesting, we just want to draw the text with a solid red brush. The XAML framework has a set of such colours built in, and so to turn the text in a `TextBox` red we just have to set the foreground brush to a solid red brush:

```
float v1 = 0;

if (!float.TryParse(firstNumberTextBox.Text, out v1))
{
    firstNumberTextBox.Foreground =
                    new SolidColorBrush(Colors.Red);
    return;
}
```

This version of the code works OK. If a user enters an invalid number (i.e. one which contains text) the colour of the foreground for that textbox is set to a solid red brush and the method ends without calculating a result. However, this is not a very good solution. The program should really check more than just the first value when it runs. If the user has typed invalid text into both boxes they would much rather find out in a single message, rather than having to fix one error and then be told later that there was also a second fault item.

There is a more serious fault too, in that the code above will turn the text red when the user enters an invalid item, but it will not turn it back to the original colour when the value is corrected. We must add an else behaviour to put the colour back to the proper brush when the value that is entered is correct. We need to store the "proper" brush so that we can use it for this. A complete version of `calculateResult` looks like this:

```
private SolidColorBrush errorBrush =
                    new SolidColorBrush(Colors.Red);
private Brush correctBrush = null;

private void calculateResult()
{
    bool errorFound = false;

    if (correctBrush == null)
        correctBrush = firstNumberTextBox.Foreground;

    float v1 = 0;

    if (!float.TryParse(firstNumberTextBox.Text, out v1)) {
        firstNumberTextBox.Foreground = errorBrush;
        errorFound = true;
    }
    else
        firstNumberTextBox.Foreground = correctBrush;

    // Repeat for v2

    if (errorFound)
        resultTextBlock.Text = "Invalid inputs";
    else
    {
        float result = v1 + v2;
        resultTextBlock.Text = result.ToString();
    }
}
```

This code turns the text red if a value is invalid. It also maintains a copy of the correct brush, so that if a value is correct it can be set to the correct colour.

This code works well and provides a reasonable user experience. It also illustrates that when you write a program with a user interface you actually end up writing a lot more code than just the statements that are needed to calculate the result. The code to deal with input errors can greatly increase the amount of code that you have to write.

4.2 Working with XAML text

XAML can be described as a 'declarative' language. This means that it doesn't actually describe any behaviour (i.e. it does not tell the computer how to do something). Instead it just tells the computer above stuff. In C# terms we tell the compiler about things we want to use (for example variables) by *declaring* them. The XAML language is only ever used to describe things; hence it is called a declarative language. We will have to get used to putting some aspects of a program into the declaration of the elements it uses. One such situation is in the configuration of the input mode for the `TextBox` elements that are used to enter the numbers in our `AddingMachine` program.

Configuring a TextBox to use the numeric keyboard

At the moment the `AddingMachine` uses the standard keyboard behaviour on the Windows Phone which is to display the text keyboard when the user selects that control. It would make sense to provide the user with a keyboard set to enter numbers, rather than the text one. There are a number of different keyboard configurations for the keyboard that a Windows Phone can display. Below you can see the standard "text" one.

Figure 4-3 Standard text keyboard

Figure 4-3 above shows the standard text keyboard. This is displayed if the user selects a TextBox in an application. To enter digits the user must press the '&123' key to switch the keyboard into numeric mode.

Figure 4-4 Numbers and punctuation keyboard

The numbers and punctuation keyboard shown in Figure 4-4 allows the user to enter digits, along with a set of punctuation characters.

It would be nice if each `TextBox` in the adding machine could be set for numeric entry when that `TextBox` is opened. We could do this by modifying the properties of the `TextBox` when the program starts running, but the best place to put such settings is in the XAML description of the control itself. At the moment the XAML that describes the `secondNumberTextBox` is as follows:

```
<TextBox x:Name="secondNumberTextBox" HorizontalAlignment="Left" Height="72" Margin="0,20,0,0"
TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top" Width="456"/>
```

This tells the window manager (which looks after the display on a page) where on the screen to put the element, the size of the element and the name it has. There is also some text alignment information.

Each of the values enclosed in `<TextBox .. />` is an *attribute* of the textbox. Attributes are simple items that are not structured and can be expressed as a name – value pair. Above you can see that the `Height` of the element, the `Name` of the element and lots of the other properties that are expressed in this way.

To express this we could use a slightly more complex version of `TextBox`:

```
<TextBox x:Name="secondNumberTextBox" HorizontalAlignment="Left" Height="72"
Margin="0,20,0,0" TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top"
Width="456">
    <TextBox.InputScope>
        <InputScope>
            <InputScopeName NameValue="Digits" />
        </InputScope>
    </TextBox.InputScope>
</TextBox>
```

Good thing I said “slightly” more complex, eh? If you want a XAML element to contain properties you have to use a different form of the element description. This is a bit confusing in the XAML context, so we might want to take a look at something a bit easier to understand. The good news is that we can simplify the text above a lot, but before we do this it is useful to understand something about the structure of XAML and the XML language it is based on.

How elements and properties work together in XAML and XML

Remember that XAML is based on XML? The XML (eXtensible Markup Language) standard allows us to design languages that can describe anything. At this point it is worth spending some time discovering how this works. We can use the XML format to create a language that will hold information about people. We could start by just storing the name of a person:

```
<Person name="Rob"/>
```

This element is called “Person” and has a single attribute called “Name” which is set to “Rob”. The XML compiler sees the `/>` at the end of the element definition and knows that it has reached the end of the description of this element.

Element with properties

However, we might want to store more complicated information about a person, such as where they live. Their address will be made up of lots of different elements, and so we should make this address a property of the person.

```
<Person name="Rob">
      <Address addressType="HomeAddress">
            <FirstLine text="18 Pussycat Mews"/>
            <Town text="Seldom"/>
            <County text="Wilts"/>
            <PostCode text="NE1 410S"/>
      </Address>
</Person>
```

This element contains one property, which is the address of the person. The address itself is another element which contains a single attribute (the type of the address) and then four properties. This illustrates an important point with XML based languages. A property is an XML element like any other, which means that it can have attributes and contain properties of its own. XAML uses this to good effect with the `Grid` element, which can hold a collection of elements (including Grids) which are to be laid out on the page.

The XAML compiler can tell that an element contains properties because the first line of the element description ends with `>` rather than ending `/>`. The XAML compiler will regard everything it sees after the first line as a property of the element, until it sees the `</ElementName>` which rounds things off.

If you find this confusing at first, welcome to the club. The best way to think about this is to consider what is being stored. In the case of the `Person` above the name is just a single item which can be expressed in a simple attribute. However the address will contain multiple properties of its own and so is best made into a property. There is also the possibility that a `Person` will have several address values, for example a `HomeAddress` and a `WorkAddress` and so the above design makes it possible to add new address types easily.

The type of the address has been made an *attribute* of the address, which I think is the most appropriate. Whether to make a data item an attribute or a property is something you have to think about when you use XML (eXtensible Markup Language) to describe things.

If we go back to our `TextBox` we should find that things make a bit more sense now.

```
<TextBox Height="72" HorizontalAlignment="Left" Margin="8,19,0,0"
Name="firstNumberTextBox" Text="0" VerticalAlignment="Top" Width="460"
TextAlignment="Center">
    <TextBox.InputScope>
        <InputScope>
            <InputScopeName NameValue="Digits" />
        </InputScope>
    </TextBox.InputScope>
</TextBox>
```

Simple information about the `TextBox` is given as attributes, whereas the `InputScope` value is given as a property. The `InputScope` itself contains a set of properties, each of which describes a particular input scope. This means that one day we might be able to ask for an input scope that includes `Digits` and `Text`, but not punctuation. However, we don't want anything as complex as that. We just want to ask for Digit input, and so that is the `InputScope` that we set up. If we use the above description for the two textboxes in the adding machine we get a vastly improved user interface where the user is given the numeric keyboard when they move to those input elements.

This is an example of a situation where just adding text to the XAML is the quickest and easiest way to configure the input behaviour. We could have used the Visual Studio property pane, or even written some complex code to build a collection of `InputScopeName` values and assign them to the `TextBox`, but this solution is much neater, and also allowed us to refine our understanding of XAML.

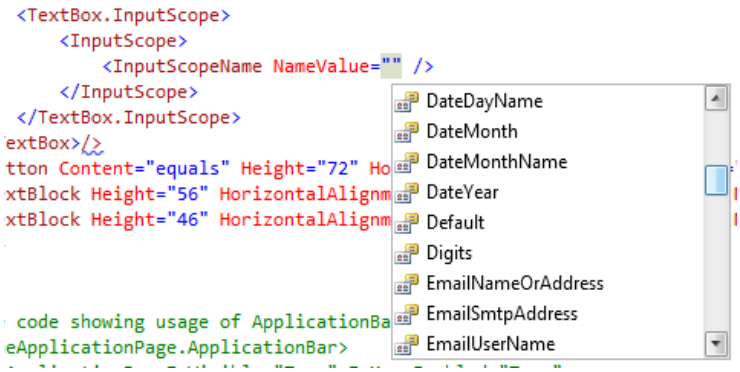


Figure 4-5 Intellisense when editing XAML

Visual Studio provides a feature called "Intellisense" where the editor tries to give you help by figuring out what you are trying to type. Because Visual Studio knows all the possible values for an `InputScope` item it suggests them when your start to enter that expression. Above you can see what happens when Intellisense detects that we are entering an InputScope value.

XAML Attributes and Properties – the truth

It turns out that from the point of view of XAML attributes and properties are actually interchangeable. The software that uses the XML description can get information from either parts of an XML element. Rather than the somewhat complicated format above we can actually write:

```
<TextBox InputScope="Number" Height="72" HorizontalAlignment="Left"
Margin="144,44,0,0"
Name="startHourTextBox" Text="00" VerticalAlignment="Top" Width="104"
TextAlignment="Center" />
```

This sets the input scope of the `TextBox` to numbers. It doesn't work if we want to set multiple scope values at the same time, but it turns out that it is only useful to set one value anyway. The reverse of this is also true.

```
<TextBox HorizontalAlignment="Left" Margin="8,175,0,0" Name="secondNumberTextBox"
Text="0" VerticalAlignment="Top" Width="460" TextAlignment="Center">
    <TextBox.Height>
        72
    </TextBox.Height>
</TextBox>
```

In the XAML above the height of the `TextBox` has been brought out of the attributes and turned into a property. The property name must contain the name of the element type that it is part of. We can't just say `Height`, we have to put

`TextBox.Height`. This is not a restriction of XML (we didn't have to put this in our Person example above) but is a requirement for XAML.

From the point of view of an XAML writer it doesn't really matter which of these forms we use. For simple name/value pairs we can use attributes, but for more complicated structures we may decide to use properties.

C# Property Setting

If you were wondering what the C# to do the same setting would look like, this is it:

```
// Make a new input scope
InputScope digitScope = new InputScope();

// Make a new input scope name
InputScopeName digits = new InputScopeName();
// Set the new name to Digits
digits.NameValue = InputScopeNameValue.Digits;

// Add the name to the new scope
digitScope.Names.Add(digits);

// Set the scope of the textbox to the new scope
firstNumberTextBox.InputScope = digitScope;
```

If you read carefully through this code you will see that it is actually building up each of the elements in the XAML above.

The solution in *Demo 01 AddingMachine with Error Checking* implements a version of the AddingMachine that performs validation and turns invalid entries red. It also sets the input scope of each TextBox to numeric.

Displaying a MessageBox

There is a `MessageBox` object which you can use to inform the user of errors:

```
MessageBox.Show("Invalid Input");
```

This will display a message at the top of the phone screen that the user must clear before they can continue. An alert sound is also played.

Figure 4-6 Single line message

If you want to send larger messages to the user of the phone you can do this by displaying several lines in the message:

```
MessageBox.Show("Invalid Input" +
    System.Environment.NewLine +
    "Please re-enter");
```

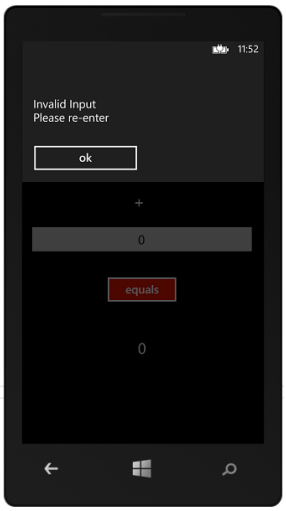


Figure 4-7 Multi-line messages

The element `System.Environment.NewLine` will provide a newline character in a manner appropriate to the target platform. A `MessageBox` is useful if you want to be sure that the user has acknowledged a message before continuing. However, you need to remember that when it is displayed it will stop the program at that point.

The solution in *Demo 02 AddingMachine with MessageBox* displays a message box when the user enters an invalid value.

Message Box with selection

We can ask the user to make a choice in a `MessageBox`:

```
if (MessageBox.Show("Do you really want to do this?",
                    "Scary Thing", MessageBoxButton.OKCancel)
    == MessageBoxResult.OK)
{
    // do scary thing here
}
else
{
    // do something else
}
```

This version displays a message with a cancel option as shown below. The if condition is evaluated according to the response from the user.

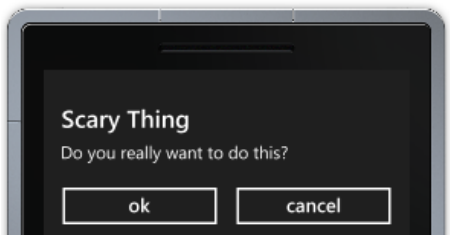


Figure 4-8 Message Box with confirmation

Adding and Using Graphical Assets

In the previous section we saw how Visual Studio manages assets such as images and sounds and adds these to the XAP file for transfer into the target device. Now we are going to see how our applications can add assets of our own and then use it.

A background screen for the Adding Machine

Figure 4-9 Some gears in a difference engine

We have been asked by the son of the boss of our customer (who seems to think he is a graphic designer) to add the above image as a background to the adding machine application. This is actually a photograph of some of the gears of Charles Babbage's "Difference Engine" and he thinks that this addition will make the program much better. We're not convinced, but as he is the son of the boss we have to go along with it.

Handling different types of displays

When you add assets such as this it is important to remember the display resolution of the target device. There are a variety of different screen resolutions for a Windows Phone device, and you need to make sure that your assets look good on all the versions of the platform. These are the supported sizes at the moment:

WVGA	480 × 800	15:9 aspect ratio (this is the Windows Phone 7 size)
WXGA	768 × 1280	15:9 aspect ratio
720p	720 × 1280	16:9 aspect ratio

It is important to note that the highest resolution screen (720p) also has a different *aspect ratio* to the other phone types. The aspect ratio of a screen is the ratio of the width to the height. Old style TV screens had a ratio of 4:3, i.e. 4 units across and 3 down. This gave them a fairly square appearance. Widescreen TVs have an aspect ratio of 16:9, where the height is slightly more than half the width, leading to a much wider looking display. Hence the name I suppose.

If you display an image without regard to the aspect ratio of the original and that of the output display it might look wrong. Circles will become ellipses and images of people will look thinner or fatter. This may or may not be a problem, but the important thing to make sure is that you check that things look OK. You can select emulators that use the different types of display from within Visual Studio.

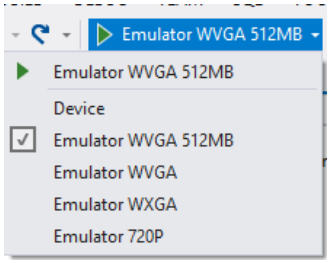


Figure 4-10 Selecting the emulator

There may be new devices coming with different screen sizes in the future, but you can use the menu shown in Figure 4.9 to choose between them. For the purpose of our artwork we have decided that the difference in aspect ratio of the 720p

display will not have an adverse effect on the appearance of the program display, so we are going to fill the entire screen with the background.

Adding Images as Items of Content

We can add an image as an item of content to the `AddingMachine` project. The boss's son has given us a jpeg image called `AddingGears.jpg` to use. It turns out that the quickest way to add a content item to a project in Visual Studio is just to drag the item out of the folder and drop it onto the project:

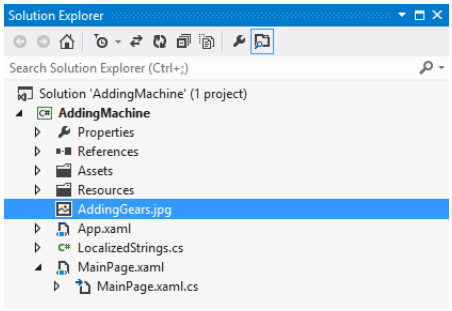


Figure 4-11Adding a resource to a project

This makes a copy of the resource file in the project directory at the appropriate point.

Links to assets

If we want to share an asset amongst a number of different projects (perhaps you have a company logo that is to be used by lots of different programs) then we can add a link to a resource, but to do this we have to add the item using the `Add Existing Item` dialog and then select "`Add as Link`":

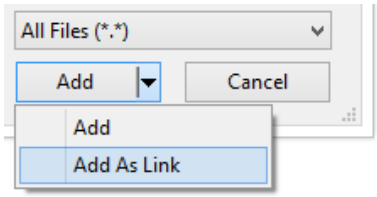


Figure 4-12 Adding a link to a resource

Figure 4-12 shows how you manage the way that content is added to a project. The Add and Cancel buttons are at the bottom left of the file dialog. If we add a file as a link and then we change the logo file in its original location the new version will be picked up by projects that use it next time they are rebuilt.

The "Build Action" of an Asset

When Visual Studio builds a project there are a number of ways that assets can be added to the completed program. We are going to consider two; adding the asset as an item of content or adding the asset as a resource inside the program assembly. The decision of which to use can have a bearing on the way that programs load and run, so it is worth knowing a bit about how this works.

Adding an asset as Content

Visual Studio needs to know what to do with each asset that is added to the project. In the case of the image we want the image file to be available when the program runs. We want it to be a piece of *content* that is copied into the program folder when the solution is deployed. We manage this from the Properties pane for the content item. When an asset is added to a project it is initially added with a build action of "Content", but Visual Studio will **not** copy this file into the application folder:

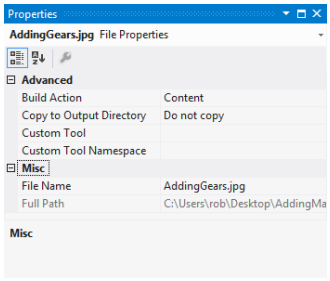


Figure 4-13 Initial properties for a content item

Figure 4-13 shows the initial properties for an item of content. You can open the properties pane by right clicking on an item in the Solution Explorer and then selecting "Properties" from the context menu that appears. The settings shown in Figure 4-12 are almost right, but we do actually want to make a copy of the content so that our program can use it. This means that the "Copy to Output Directory" behaviour needs to be changed.

If we want to use the asset as content we need to change its properties so that it is an item of content. We do this by changing the Build Action entry for the item of content.

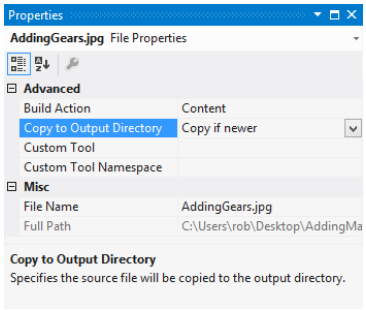


Figure 4-14 Setting the correct copy behavior

Figure 4-14 shows the settings that are required. The "Copy if newer" selection means that Visual Studio will copy a new version of the image into the binary folder if the current image in the project is replaced by a newer file.

Using image content in a program

An asset of type content is simply made available to the application as a file in the same folder as the program itself. To display an image held as content in our program we just need to add the line of XAML that describes the image file that we want to use:

```
<Image HorizontalAlignment="Left" Margin="0,0,0,0" Grid.Row="1"
VerticalAlignment="Top" Stretch="Fill" Source="AddingGears.jpg"/>
```

The `Image` element displays an image on the page. It has a `Source` attribute that identifies the file from which the image is to be loaded. If we enter a filename as the source the image will be loaded from the file. When the program runs the file will be loaded from the file and then drawn on the screen. The program draws the elements on the screen in the order they are defined in the XAML. If this line is given before the `Grid` containing the elements the image will be drawn in the background:

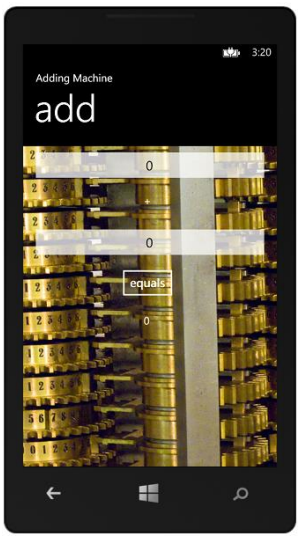


Figure 4-15 Displaying a background image

This is the result of the program. From an artistic point of view we are not convinced, by the choice of picture but the process itself works.

The solution in *Demo 03 AddingMachine with Background Content* displays a background image for the application which is loaded as an item of content.

Rather than modifying the XAML directly, we can set the source image from the properties of the image item.

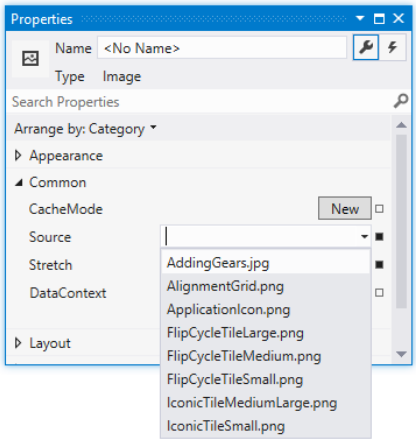


Figure 4-16 Selecting the source of an image

Figure 4-16 shows where on the property item you would do this. The `ListBox` selector on the source of an image shows you all the content images that are available. The rest of the images are added to the project when it is built.

Making content available to the application

When Visual Studio builds the application it makes sure that the content items are available **as long as it has been instructed to copy them into place.** If you forget to select this option your program will fail when it runs.

4.3 Using the TextChanged Event

Our adding machine is now quite useable, but the customer is still not completely happy. He doesn't like the way that users have to press the Calculate button each time they want a new answer. What he wants is for the program to detect when the text in the number boxes changes and then update the total automatically. Fortunately for us this turns out to be quite easy to do. We need to revisit events to find out more.

We first saw events when we considered the `Button` element. A `Button` element generates an event when it is activated. From a program point of view an event is a call to a method. We can get Visual Studio to "hook up" a button to an event

for us just by double clicking the button in the designer. At the moment the `Calculate` button has a click behaviour that calculates and displays the result. The method to be called is identified in the `Button` description:

```
<Button x:Name="equalsButton" Content="equals" HorizontalAlignment="Left"
Margin="165,223,0,0" VerticalAlignment="Top" Click="equalsButton_Click"/>
```

When the button is clicked this causes the nominated event handler to run inside the page class. In the case of our `AddingMachine` this will call the `calculateResult` method.

```
private void equalsButton_Click(object sender, RoutedEventArgs e)
{
    calculateResult();
}
```

This is fine for buttons, but we want something similar to happen when the text in a `TextBox` is changed by the user of the program. To achieve this we need to find an event that will deliver this message. We can find out what events a `TextBox` can create by selecting a `TextBox` in the Visual Studio editor and then selecting `Events` in the properties pane.

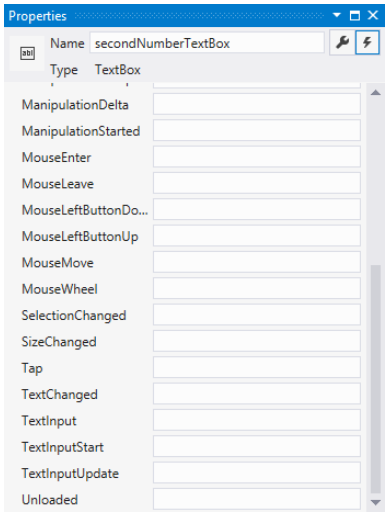


Figure 4-17The TextBox events

A `TextBox` can produce a lot of events, including some which seem somewhat meaningless on a Windows Phone. We are going to use the `TextChanged` event. This is fired each time the text in the `TextBox` is changed. If we double click on the `TextChanged` area in the Properties pane for a `TextBox` Visual Studio will create a method and hook it up to this event.

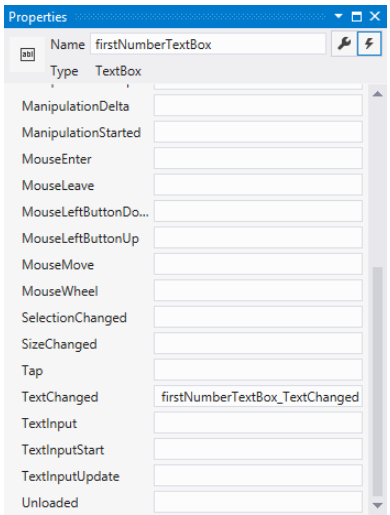


Figure 4-18 The TextChanged event handler

Figure 4-18 shows the event hander. The actual code for the handler method is placed in the `MainPage.xaml.cs` source file:

```
private void firstNumberTextBox_TextChanged(object sender,
                                            TextChangedEventArgs e)
{

}
```

We just need to add a call of `calculateResult` into this method:

```
private void firstNumberTextBox_TextChanged(object sender,
                                            TextChangedEventArgs e)
{
     calculateResult();
}
```

We can do the same thing for the second text box, so that if either of the boxes is changed the sum is updated. This means we can now remove the Calculate button.

The solution in *Demo 4 AddingMachine with no button* implements an adding machine which uses TextChanged events and does not have a calculate button.

One rather amusing thing that might trip you up is that the changed events are also fired when a program makes a change to a display component.

```
firstNumberTextBox.Text = "99";
```

The statement above puts the string 99 in the textbox. This would cause the changed event to fire (which is exactly what should happen). However, you can get yourself into trouble if the code that runs in the event of a change to the content of a box also changes the context of that box. In other words, if the changed event handler contained a statement like the one above the program would be stuck in an "event loop" with events firing, causing other events and so on for ever.

4.4 Managing Application Page Layout

You might have noticed that there are two ways that users can hold the device, landscape (device on its side) and Portrait (device held upright). The phone itself is able to detect the way it is being held and some built in applications respond correctly when the phone is tipped.

Unfortunately our adding machine doesn't handle this very well:

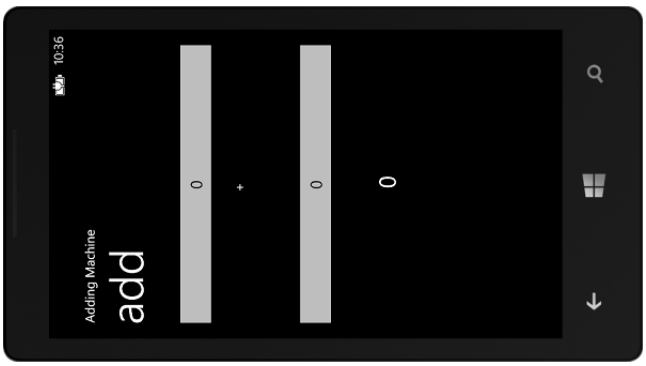


Figure 4-19 Landscape mode and the Adding Machine

Now, there is nothing wrong with only working in one orientation, the Start Menu in Windows Phone only works if the phone is held in the portrait orientation. But remember that if you decide to support both ways of holding the phone you will have to spend twice as much time designing the user interface and then add the code to respond when the user tilts the phone. The bottom line (for me at least) is that if your customer says "You know, it would be great if the program worked when the phone was held on its side" you don't immediately say "I can do that". Instead you should say "That will cost extra". Because it will. However, if you have been offered big bucks to add this feature, it is useful to know the best way to do this.

Landscape and Portrait Selection in MainPage.XAML

A given application can hold lots of different pages. Later we will see how to create these and navigate between them. Each page in an application can support one or more orientations, depending on what the application wants to use it for. A page can state the orientations that it supports. This is expressed as a property of the PhoneApplicationPage:

```
SupportedOrientations="Portrait" Orientation="Portrait"
```

The settings that you see above are the ones that Visual Studio puts in the page when it is created. This means that the page only work in Portrait mode and its initial setting is Portrait, which makes sense. We can modify this if we want to:

```
SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
```

This is a bit of an improvement, that when the program runs we can now tip the screen and see this:

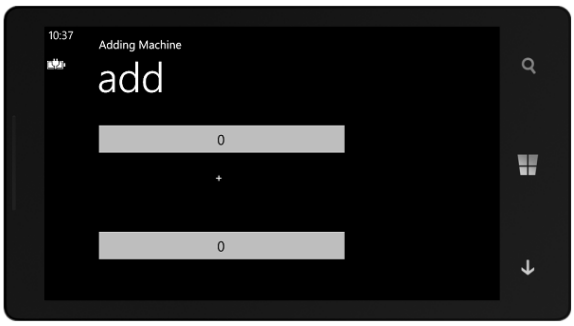


Figure 4-20 Landscape mode fail

The technical term for this is "Close, but no cigar". Our program now redraws the screen when the orientation changes, but because all the components are positioned absolutely on a grid, some of them are in the wrong place, and the answer doesn't fit on the screen any more.

```
<TextBox Height="72" HorizontalAlignment="Left" Margin="8,19,0,0"
   Name="firstNumberTextBox" Text="0" VerticalAlignment="Top" Width="460"
   TextAlignment="Center" />
<Button Content="equals" Height="72" HorizontalAlignment="Left"
   Margin="158,275,0,0" Name="equalsButton" VerticalAlignment="Top" Width="160"
   Click="equalsButton_Click" />
```

These two components have `Margin` values that are used to precisely position them on the screen. Unfortunately, some of the positions are off the bottom of the screen. The system will not complain about this, but the components won't be displayed either.

There are two ways we can fix this. One way is to provide a completely different layout for the landscape screen and switch between the two layouts when the orientation changes.

Orientation Changed Event

We have already seen how display elements can generate events. We attached an event to the Click event generated by the `Button` that the user presses when they want the result of a calculation. We can also attach event handlers to other events too. The `PhoneApplicationPage` can generate a whole set of events, including one which fires when the orientation of the page changes.

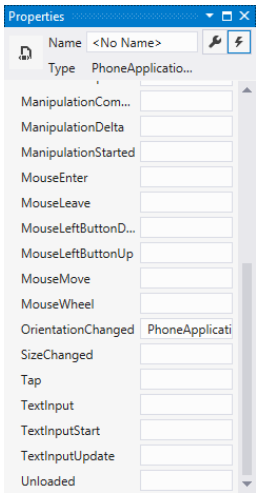


Figure 4-21OrientationChanged event handler

Above you can see how we can add a method to this event. When the orientation of the phone changes this method will be called to adjust the display. We just have to add some code that does this:

```
private void PhoneApplicationPage_OrientationChanged(
                    object sender, OrientationChangedEventArgs e)
{
    if (e.Orientation == PageOrientation.PortraitUp ||
        e.Orientation == PageOrientation.PortraitDown)
    {
        setPortrait();
    }
    else
    {
        setLandscape();
    }
}
```

This code uses the parameter to the method to detect the new orientation and call the appropriate method. The methods modify the margins of the controls to lay them out properly:

```
private void setLandscape()
{
    firstNumberTextBox.Margin = new Thickness(8, 19, 0, 0);
    firstNumberTextBox.Width = 207;
    secondNumberTextBox.Margin = new Thickness(252, 19, 0, 0);
    secondNumberTextBox.Width = 207;
    plusTextBlock.Margin = new Thickness(221, 35, 0, 0);
    resultTextBlock.Margin = new Thickness(538, 35, 0, 0);
}
```

This is the `setLandscape` method. The `setPortrait` method is very similar. Controls are positioned by setting a `Margin` around them. The margin of a control is given by a `Thickness` value which has margin size and a border size. We don't want to actually draw any borders for these controls, and so the border size values are set to 0. If I add these methods to my new application it now works fine in either mode:

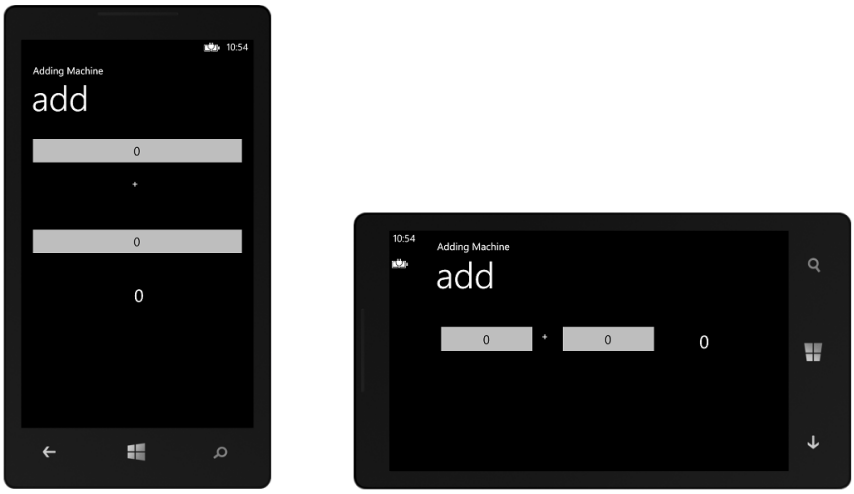


Figure 4-22 Orientation Changes

Figure 4-22 shows the two different orientations in action. When I moved the phone into landscape mode the `OrientationChanged` event fired and the layout was updated. However, and this is important, if the program is started and the phone is already being held in portrait orientation the system will not generate an orientation changed event. This means that the settings for your display elements and the values in the `setPortrait` method must align, otherwise the user will see a different display when they start the program with the phone held in portrait mode.

We can fix this by making a method that does the alignment and then calling this method when the program starts, as well as when the orientation changes:

```
private void fixOrientation()
{
    if (this.Orientation == PageOrientation.LandscapeLeft ||
        this.Orientation==PageOrientation.LandscapeRight)
    {
        setLandscape();
    }
    else
    {
        setPortrait();
    }
}
```

This method uses the `Orientation` property of a page to decide which alignment method to call.

```
private void PhoneApplicationPage_OrientationChanged(
                    object sender, OrientationChangedEventArgs e)
{
    fixOrientation ();
}
```

The last thing we need to do is add a call to the `fixOrientation` method when the page is displayed to the user. We can do this by overriding a method on the page:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    fixOrientation();
}
```

The `OnNavigatedTo` method is called by the system when the user "arrives" at a page. It is a very useful place to get control and set up a display before the user sees it. The code above overrides the method in the parent of the page and adds a call of `fixOrientation`. Note that it also calls the method in the parent.

Screen Resolution Issues

Figure 4-23 WVGA and WXGA displays

The Windows Phone platform is available with a number of different sized screens. Figure 4-23 shows the Adder application running on devices with two different sized displays. You might find this confusing, as you may expect that items on the larger device to be in the "wrong" position, since we positioned the components using specific screen coordinates.

Windows phone handles this by automatically adding a "multiplication factor" to coordinates for larger screens. This is applied to position values so that everything appears in the right place. A program can use the WVGA coordinate system and this will work on all the platforms. The actual screen dimensions and multiplication factors are as follows:

WVGA	480 × 800	15:9 aspect ratio – multiplication factor of 1
WXGA	768 × 1280	15:9 aspect ratio – multiplication factor of 1.6
720p	720 × 1280	16:9 aspect ratio – multiplication factor of 1.5, and 53 extra pixels height

Note that this scaling is applied to text on the screen as well, which means that the letters on the higher resolution screens will occupy the same amount of space on the screen, but they will be of higher quality on the high-resolution screens.

One thing to beware of is that the 720p display size has a different aspect ratio, which means that a program has an extra 53 pixels in the long dimension.

Programmer's Point: Test your program on all screen sizes

While the scaling described above will help to make sure that programs look the same on all devices, there is no substitute for running the program on a device with the given screen size. You should therefore try running your program on all the emulators before you release it.

The solution in *Demo 05 AddingMachine with auto orientation* implements an adding machine which automatically handles orientation changes.

Using Containers to Layout displays

We now know how to make applications that adjust the properties of the controls under program control. However, we still have to come up with the values to make the layouts work. Fortunately XAML can help us with this as well; in fact it can even automatically lay out an entire display for us. This can save a lot of work. The key to understanding how this works is to take a look at how XAML components can be used to hold other items. There are a number of different container types in XAML. We are going to use the `StackPanel` container.

A `StackPanel` contains a stack of display elements. It lays them out in the order that they are listed inside the panel. This can be vertical (down the display) or horizontal (across the display). Rather than giving each of our display elements a margin that positions them on the screen, we can instead get a `StackPanel` to do the layout for us:

```
<StackPanel>
    <TextBox InputScope="Digits" Height="72" HorizontalAlignment="Center"
       Name="firstNumberTextBox" VerticalAlignment="Top" Width="460"
       TextAlignment="Center" />
    <TextBlock Height="56" HorizontalAlignment="Center" Name="plusTextBlock"
       Text="+" VerticalAlignment="Top" FontSize="32" Width="25" />
    <TextBox InputScope="Digits" Height="72" HorizontalAlignment="Center"
       Name="secondNumberTextBox" VerticalAlignment="Top" Width="460"
       TextAlignment="Center"/>
    <TextBlock Height="46" HorizontalAlignment="Center" Name="resultTextBlock"
       VerticalAlignment="Top" FontSize="30" Width="160" TextAlignment="Center"
/>
</StackPanel>
```

The great thing about this is that we don't have decide where on the screen each item needs to go, they are just stacked on top of each other. I've changed the alignment of each of the elements to "Center" so that if the `StackPanel` gives an item a particular area of the screen it will centralise itself in that area. When we run the program with the layout you can see below:

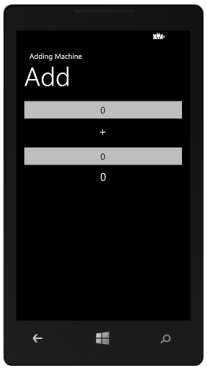


Figure 4-24 Layout using a StackPanel

Each of the items is displayed in a stack. One very useful side effect of this approach is that the `StackPanel` will re-arrange the elements if the screen dimensions change. This means that if the orientation of the phone is changed it automatically adjusts:

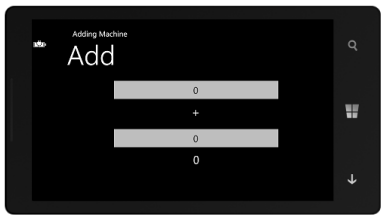


Figure 4-25 Changing orientation using StackPanel

Note that because the items are centered they now draw themselves in the middle of the landscape screen.

We can add a `StackPanel` to a display design by dragging it from the Visual Studio Toolbox onto the design surface or by typing the XAML text by hand. If we want to make a horizontal `StackPanel` we just have to add an additional attribute:

```
<StackPanel Orientation="Horizontal">
```

We can put one `StackPanel` inside another to make complex displays, as we shall see later.

There are other container types available. One type, which is used in the Windows Phone pages themselves, is the `Grid`. This allows you to create a rectangular grid of elements, each of which contains one or more display items positioned within it. In fact the Windows Phone program display that we have been using up to now has been one element in the grid that makes up the entire display area.

The solution in *Demo 06 AddingMachine with StackPanel* implements an adding machine which uses a StackPanel to automatically handle orientation changes.

Programmer's Point: You can use the StackPanel to help with orientation issues

It is sometimes hard to manage the change from landscape to portrait orientation. One trick that might help is to break the screen into regions, each of which is a StackPanel. Then, when the phone is moved from portrait to landscape you move the stack panels so that they are adjacent, rather than on top of each other. The StackPanel can automatically handle the precise positioning of the elements, and your program just has to move the panels.

4.5 Data Binding

Data Binding makes it much easier to write programs that have user interfaces. At the moment our Adding Machine works very well, in that changes made by the user to numbers are reflected instantly in the result. But we had to write the code to do all the hard work. We can make things a lot simpler by taking a look at *Data Binding*.

Data binding takes the place of the "glue" code that we are writing to connect display elements to the methods that work on them. It allows us to take a property of a display element and connect it directly to an object. All we have to do is create an object that behaves in a particular way and we can get values in the object transferred into the display element with no effort on our part.

Data binding can work in either direction. We can bind the text a `TextBox` to our application so that when a user changes the number in one of the inputs to the adding machine the program is notified of this change so that the result can be updated. We can also bind the output element to the display so that when the program changes the result value the display update occurs without us having to do anything.

We are going to create a class called `Adder` which will provide the required behaviours for an adding machine. It will do this in a manner which is completely decoupled from the user interface, which will make it easy to test programmatically, i.e. we could create a program that will test the `Adder` class.

The `Adder` class interacts with its environment in three ways:

1. It must be told when the first value (V1) changes.
2. It must be told when the second value (V2) changes.
3. It must tell you, when asked, what the current result is.

An adding machine class could be as follows:

```
public class Adder
{
    private int v1Value;

    public int V1
    {
        get
        {
            return v1Value;
        }
        set
        {
            v1Value = value;
        }
    }

    private int v2Value;

    public int V2
    {
        get
        {
            return v2Value;
        }
        set
        {
            v2Value = value;

        }
    }

    public int Answer
    {
        get
        {
            return v1Value + v2Value;
        }
    }
}
```

It exposes properties which allow the outside world to interact with it. I can test the class as follows:

```
int failCount = 0;
Adder a = new Adder();
a.V1 = 2;
a.V2 = 3;
if (a.Answer != 5)
{
    failCount = failCount + 1;
}
```

The code above creates and `Adder` instance and then checks to see if it can add 2 and 3 to produce 5. If it fails the test the program increases a fail counter by 1. If this was a production piece of code, and the `Adder` class was doing some proper data processing there would be many such tests.

Programmer's Point: Take a look at Test Driven Development

You can uses a mechanism called *assert* to integrate these kinds of tests into your programs. If you do this properly you can build a whole set of automatic tests which can be run at the push of a button, and give individual feedback on the tests that were passed and failed.

Adding Data Binding

At the moment a program can interact with the `Adder` class by using the three properties that it exposes. The idea behind data binding is that the display elements will be *bound* to the properties in the class, so that changes made by the user to data in the interface will be reflected back into the class which will process the data.

We have already used a primitive form of data binding; we configured the `TextBox` so that when the user changed a value in the `TextBox` an event is fired in our program. Now we want to make the process more automatic.

If this is confusing, think about what is going to happen when the program is running:

1. The user enters a new value into the textbox which is bound to V1.
2. The data binding system then changes the property of V1 in the Adder class.
3. At this point the answer display needs to be updated, so the Adder class must have a way of indicating that the Answer has changed, and so its display must be updated. The Adder class generates an event to tell the display environment that this value must be updated.
4. The display environment than reads the Answer property from the Adder to get an updated value for display.
5. The user notices that the answer has changed to reflect the new input and is completely unimpressed, because that is what the program is supposed to do.

Data binding is not just used for text inputs from the user, it can also be used to bind things like sliders and radio buttons on the display. In all the cases though, the principle is that you "bind" the control to a property in a class which is then changed when the content of the control changes.

Delivering new values to a databound class is easy, the external system just has to change the property:

```
a.V2 = 3
```

This tells the `Adder` instance `a` that the value of `V2` is now 3. However, the change in the value of V2 means that the value of the answer has now changed, and so the `Adder` instance has to have a way of telling the display to update the answer. It does this by causing an event in the display system. If you think about it, this is just what a Button display element does when it is pressed, except now the message is going the opposite way, from our class into the display system.

The `Button` class contains a variable that keeps track of all the objects that have registered an interest in button pressed events. In the same way, a databound class must contain a variable that keeps track of all the databound objects that need to be told when the state of the databound object changes.

```
public class Adder
{
    event PropertyChangedEventHandler PropertyChanged;
    //rest of Adder class goes here
}
```

Other objects can now connect to this event to be told when something in an `Adder` class instance has changed:

```
Adder a;
a.PropertyChanged += a_PropertyChanged;
```

The above code creates an `Adder` instance and connects a method called `a_PropertyChanged` to the changed event.

```
void a_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    //handle a change in the adder
}
```

This is the property changed event handler. We never write this code, because it is created for us automatically when we set up the data binding.

The final link in the chain is how the `Adder` class actually fires the event.

```
public int V1
{
    get
    {
        return v1Value;
    }
    set
    {
        v1Value = value;

        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs("Answer"));
        }
    }
}
```

This is the code that manages the V1 property in a databound version of the `Adder` class. When a new value of V1 is set it is necessary to update the Answer display, and so the code in the property creates a `PropertyChangedEventArgs` value that identifies this by name. Note that the code also checks to see if anyone has registered an interest in the property. If the event delegate is null there is nobody to tell.

The format of the method call that delivers our event is as follows:

```
PropertyChanged(this,new PropertyChangedEventArgs("Answer"));
```

The first parameter to the call is a reference to the object that is generating the event. The second is an argument value loaded with the name of the property that has changed. The XAML element will use reflection (i.e. it will look at the public properties exposed by `AdderClass`) to know what properties are available. If it gets a notification that `AnswerValue` has changed it will then update any display element properties that are bound to the property in this class.

Note that this is just a string of text though, if we write “answer” by mistake this will prevent the update message from being picked up correctly.

The idea is that a business class can signal which particular values need to be updated in the display. So, to recap the sequence of events in a little more detail:

1. When the program starts, the display system creates an instance of the Adder class and connects a method to the PropertyChanged event handler.
2. The user enters a new value into the textbox which is bound to V1.
3. The display system updates the property of V1 in the instance of the Adder class.
4. The property code updates the value of V1 and then calls the event handler for the PropertyChanged event, telling the display system that the “Answer” value must now be updated.
5. The display system receives the event and then goes and reads the Answer property from the Adder class, causing the result to be recalculated. It then displays the new value.
6. The user notices that the answer has changed to reflect the new input and is still completely unimpressed, because that is what the program is supposed to do.

If this is making your head hurt, and it did mine the first time I saw it, then take a deep breath and read through the sequences one more time. Keep in mind just why we are doing this. We want to put our program behaviours into separate classes so that we can develop and test them independently of the user interface, and we want to connect these classes to the interface as easily as possible. The display needs to be able to tell the program behaviours when the user has changed something, and the program behaviours need to be able to tell the display when it needs to be updated.

The INotifyPropertyChanged interface

What we have described is a perfect solution for the Adder class, but we also want to make it possible for any class to be used in the same way. We can do this by using interfaces.

In order for a class to be able to bind to objects on the display it must implement the `INotifyPropertyChanged` interface:

```
public interface INotifyPropertyChanged
{
    // Summary:
    //     Occurs when a property value changes.
    event PropertyChangedEventHandler PropertyChanged;
}
```

When a class implements an interface it is effectively saying that it "knows how to do something". In this case the interface is saying "if you implement this interface you know what to do in response to a change in a property on the screen".

Programmer's Point: Interfaces can also contain events

The first time you saw an interface it probably described some methods that a class must contain. However, it is also possible for an interface to contain an event delegate. You can think of this as a "list of people to call". Display components can add themselves to this list so that they are informed when the display needs to be updated.

The first step in using data binding with the `Adder` class is to make the class implement the interface:

```
public class Adder : INotifyPropertyChanged
{
}
```

We have added to the class definition, and told the compiler that we now implement the interface. Of course, the code above will not compile. This is because it is a broken promise. If a class says it implements an interface it must contain the elements that the interface specifies. For a class to implement the `INotifyPropertyChanged` interface it must contain the event delegate which will be used by `Adder` to tell anything that is interested when a property has changed.

```
public event PropertyChangedEventHandler PropertyChanged;
```

The final version of the class looks like this:

```
namespace AddingMachine
{
    public class Adder : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        private int v1Value;

        public int V1
        {
            get
            {
                return v1Value;
            }
            set
            {
                v1Value = value;

                if (PropertyChanged != null)
                {
                    PropertyChanged(this,
                     new PropertyChangedEventArgs("Answer"));
                }
            }
        }

        private int v2Value;

        public int V2
        {
            get
            {
                return v2Value;
            }
            set
            {
                v2Value = value;

                if (PropertyChanged != null)
                {
                    PropertyChanged(this,
                     new PropertyChangedEventArgs("Answer"));
                }
            }
        }

        public int Answer
        {
            get
            {
                return v1Value + v2Value;
            }
        }
    }
}
```

Adding the Namespace containing our class to MainPage

We now have our object that we want to data bind. The next thing we need to do is link our code to the XAML that describes our user interface. To do this we must add the namespace containing the `AdderClass` to the XAML.

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

If you look at the `AdderClass` above you will find that it is in the `AddingMachine` namespace. By adding this line at the top of the XAML file for `MainPage.xaml` we tell the system that any classes in the `AddingMachine` namespace should be made available for use in our application XAML.

In the same way that a C# program must have all the using directives at the top of the source file a XAML file must have all the namespaces that it uses at the top as well. If you look in the file at the top you will find lots of namespaces being added. These are the classes for the components that make up the user interface.

Once we have added a namespace we now have to create a name for the resources that can be found in this namespace:

```
<phone:PhoneApplicationPage.Resources>
    <local:Adder x:Key="Adder" />
</phone:PhoneApplicationPage.Resources>
```

The reason why we are doing this is that it might be useful to be able to switch the business classer around. You might make a test class that delivers sample data and behaviours and want to use this to check out the user interface. Then, later you can change to the proper one.

If this seems a lot of work then remember that you only have to do this once, and that the class that we add can contain many properties that the user interface can interact with.

Adding the Class to an component on the page

Now that the namespace is available we can connect a particular class from this namespace to the component in our page. The component we are going to add the class to will be the Grid that holds all our display components. Adding a class to a container automatically makes it available to any components inside it, and so the `TextBox` and `TextBlock` elements can all use this class.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
   DataContext="{StaticResource Adder}">
```

We get a component to use a particular class by setting a value for the `DataContext` for that component. For our application this will be a static resource, in that it will be part of the program.

Connecting the component to the property

We now have the `Adder` class available as a data context for elements on a page. This sets up a link between the page and an object that is going to provide business behaviours. The next thing we need to do is connect properties of each element to properties that our business object exposes.

We can do this from the Properties pane for an element. Let's start by connecting the text in the `firstNumberTextBox` with the `V1` property in `Adder`. If we click on the `TextBox` in the Visual Studio editor we can then open the Property pane for the item. We can then go and find the `Text` property. At the moment it is just set to a string, we want to apply a Data Binding to the Adder class. To start with, click the tiny square to the right of the `Text` item in the properties for `firstNumberTextBox`:

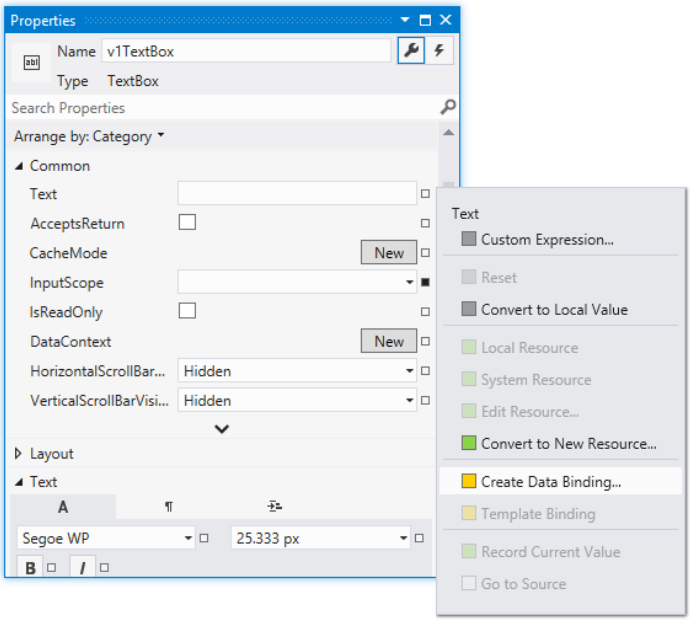


Figure 4-26 Adding Data Binding

If we select "Create Data Binding" the next thing we need to do is find the property that is going to be bound. Behind the scenes Visual Studio has gone off and opened the `Adder` class and used found out what properties it exposes. It can then display a menu of those properties that are available for data binding:

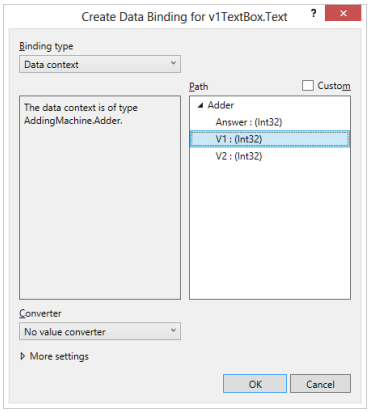


Figure 4-27 Selecting the value to bind to

This is where the magic starts to happen. Simply by selecting the required item from the list we can connect the display element property with the object.

We have to set up three bindings, one for each item on the display. The ones for `V1` and `V2` will cause corresponding property values in the `Adder` class to change when the display elements are changed. The binding for the Answer item will cause the display to be updated when the `PropertyChangedEvent` is fired.

Binding Settings

If you select "More settings" when setting up a binding the dialogue expands and you get access to additional options that can be used to configure the binding.

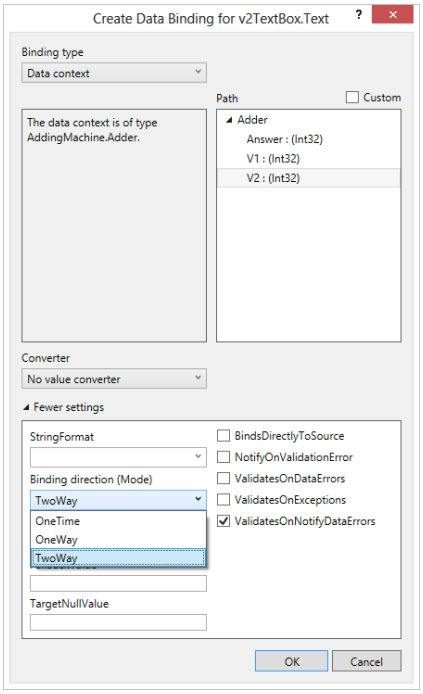


Figure 4-28 Additional Binding options

I have highlighted the Binding direction option, which you can select as appropriate:

- **OneTime** – this means that the binding is performed once at the start of the program and never again. It is useful for fixed items such as labels that the program will not change as it runs.
- **OneWay** – this means that the element on the screen is only ever written to. The Answer value is bound as a one-way binding, as it is not possible for the program to get input from a TextBox
- **TwoWay** – with this binding changes to the content of the display will cause changes to the bound object, and it is also possible to display output, as for a OneWay binding.

You can also use these settings to set up filters and display options so that the values in your objects are displayed with the correct format.

Of course, the options that you set end up as extra elements in the XAML describing the screen components:

```
<TextBlock Height="46" HorizontalAlignment="Center"
x:Name="resultTextBlock"VerticalAlignment="Top" FontSize="30"
Width="160"
TextAlignment="Center" Text="{Binding Answer, Mode=OneWay}" >
```

Above you can see the binding settings for the result text block.

Once we have made these bindings our program will just work. The interesting thing about this is that if you look in the `MainPage.xaml.cs` file for the program code that makes the application work you will find that it is completely empty. All the work is now being done by our tiny little class, which is connected to the display items. Changes to the content of the `TextBox` elements used to hold the input values cause events inside the `Adder` class. This class then changes the content of the `AnswerValue` which is bound to the `ResultTextBlock` on the display.

The solution in *Demo 07 AddingMachine with data binding* implements an adding machine which uses data binding to connect the inputs to the AdderClass.

Data Binding using the Data Context

The AddingMachine solution only ever uses one instance of the Adder class, which is created as a static resource when the program is loaded.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"
   DataContext="{StaticResource Adder}">
```

The ContentPanel element is the one that holds everything we have placed on the screen. Setting the DataContext for a container also sets it for all the items it contains, which is how the TextBlock and TextBox elements can refer to the V1, V2 and Answer properties of the Adder class.

Instead of adding the context to the XAML, as shown above, we can also set the data context for a control from within our programs. All we have to do is create an Adder instance and set the DataContext of the element that contains the firstNumberTextbox to this instance. We can do this in the constructor for the main page.

```
// Constructor
public MainPage()
{
    InitializeComponent();

    Adder adder = new Adder();
    ContentPanel.DataContext = adder;
}
```

The DataContext for an element identifies the object that contains all the properties that have been bound to by that control and any that the control contains.

The ContentPanel on the page contains our TextBox and TextBlock elements and now every binding that they contain will be mapped onto the Adder instance that we created.

The solution in *Demo 08 AddingMachine with data context* implements an adding machine which uses data binding to connect the inputs to the AdderClass.

This shows that you can design the interfaces very quickly if you write the XAML directly, rather than using the property pains. In fact, because the XAML editor is not aware of the binding that we have performed, we can't use the visual editor to set up the elements, we have to write the XAML ourselves.

Programmer's Point: Setting the Data Context is an easy way to display data

In the case of the Adding machine, there is not a lot of benefit in being able to create an Adder instance and assign it like this as you only ever need one adding machine. However, if you are selecting one item for display from a large number, this can be a very good way to do this, as all you have to do is set the Data Context and the data binding takes care of the rest. We will explore this next.

4.6 Displaying Lists of Data

Displaying single items is all very well, but we would like to be able to display lists of things. Which brings us to another problem; we need to have some things to list.

Creating a Customer List

We are going to make a little customer management program. It will be very simple; just holding the customer name, address and ID number, but we will expand it later, even going as far as putting the whole thing into an SQL database. Our first version of the customer class looks like this:

```
public class Customer
{
    public string Name { get; set; }
    public string Address { get; set; }
    public int ID { get; set; }

    public Customer(string inName, string inAddress, int inID)
    {
        Name = inName;
        Address = inAddress;
        ID = inID;
    }
}
```

I have implemented the data fields as properties, we will see why this is a good idea later. The `Customer` class just contains a single method, the constructor, which creates an instance of the class from three input values.

Now we need to store a list of customers. The best way to store lists of things is to use the `List` collection class provided by C#. We can create a `Customers` class that contains the name of the customer list as a string and the list of customers:

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

    public List<Customer> CustomerList;

    public Customers(string inName)
    {
        Name = inName;
        CustomerList = new List<Customer>();
    }
}
```

The `Customers` class only contains a single method which is the constructor for the class. This sets the name of the class and then creates an empty list to store them in. If I wanted to create a customer list for my mail order customers I could write it like this:

```
Customers mailCustomers = new Customers("Mail Order Customers");
```

We now have two classes, one which can hold a single customer and one that can hold a large number of them.

Making Sample Data

The next thing we want is a whole bunch of customers. We could type these in by hand, and spend many happy hours inventing interesting names, but this would be silly. A better way of doing this is to create a method to do the hard work for us. We can put the method in the `Customers` class so that we can ask the class to give us some test data:

```
public static Customers MakeTestCustomers()
{
    string  [] firstNames = new string [] { "Rob", "Jim", "Joe",
                                            "Nigel", "Sally", "Tim"} ;
    string[] lastsNames = new string[] { "Smith", "Jones",
                  "Bloggs", "Miles", "Wilkinson", "Brown"

 };
    Customers result = new Customers("Test Customers");
    int id = 0;

    foreach (string lastName in lastsNames)
    {
        foreach (string firstname in firstNames)
        {
            //Construct a customer name
            string name = firstname + " " + lastName;
            //Add the new customer to the list
            result.CustomerList.Add(new Customer(name,
                                        name + "'s House", id));
            // Increase the ID for the next customer
            id++;
        }
    }
    return result;
}
```

This code will create 36 customers by combining all the first and last names that are given. If we want many hundreds of customers we just have to add more name strings. Each customer has a unique ID number. The `MakeTestCustomers` method has been made static, so that we don't need to have a customers list before we can create the test data. The customer list that is created is called "Test Customers". We can get our list very simply:

```
customerList = Customers.MakeTestCustomers();
```

Whenever I make something that is supposed to hold a large number of items the next thing I do is create a quick way of getting a bunch of them. If you are worried about this affecting the size of the program you can use conditional compilation so that the `MakeTestCustomers` method is only included in the class if a particular debug symbol is defined when the program is built.

Programmer's Point: Make test data as soon as you can

I hope that this illustrates a very important principle. As soon as you make something you need to create a way of testing that thing and proving it works well. If we were selling our customer management system to a shop owner she would not be impressed if we showed it working with only a couple of customers that we had entered by hand. She would want to see the program managing many thousands of entries. It is not that hard to create code that does this, as you see above, and it makes the testing of the program much easier, and more convincing.

Using the StackPanel to display a List

The first thing we are going to do with our customer list is just display all of the customer names on the phone screen. We can put each name into a `TextBlock`. We've created `TextBlock` elements before; they are how our adding machine displays the answer to the user. What we want is something that will lay out the textboxes for us automatically, even if we don't know just how many there will be when the program runs. We know how to do this as well. We can use a `StackPanel` to lay out a large number of elements automatically. This is what the XAML for the display looks like:

```
<StackPanel HorizontalAlignment="Left"
  Margin="0,0,0,0" Name="customersStackPanel"
  VerticalAlignment="Top"/>
```

You may be wondering where all the `TextBlock` values are? The answer is that we can create them in the program:

```
customers = Customers.MakeTestCustomers();
foreach (Customer c in customers.CustomerList)
{
    TextBlock customerBlock = new TextBlock();
    customerBlock.Text = c.Name;
    customersStackPanel.Children.Add(customerBlock);
}
```

This code creates a set of test customers and then works through them. For each customer it creates a new `TextBlock` instance called `customerBlock`. It then sets the `Text` property of `customerBlock` to the name of the customer. Finally it adds `customerBlock` to the children of `customersStackPanel`.

Programmer's Point: Your program can create display

This illustrates a very important point. We have seen that you can create display elements by describing them in the XAML for a page. However, we can also create display elements by constructing them as objects in our programs. If you don't know how many elements you need, just make them in the program and add them to a container.

Figure 4-29 A list of customers in a StackPanel

Above you can see how the `StackPanel` has displayed a stack of customer names. The above program appears to work well for any number of customers. However, we have a problem. There are some customers below Nigel Miles, but we can't see them because they have "fallen off" the screen. The height of a `StackPanel` will grow to accommodate the items that are placed in it, but they might not all fit on the display.

The display will not produce an error if we try to draw something that won't fit on the screen, but the display won't look right. We need to find a way of allowing the user to scroll the display up and down to see the rest of the list. We can do this with a `ScrollViewer`. This provides a scrollable window which lets the user pan around a displayed item.

```
<ScrollViewer>
    <StackPanel HorizontalAlignment="Left" Margin="9,6,0,0"
                Name="customersStackPanel" VerticalAlignment="Top"/>
</ScrollViewer>
```

Figure 4-30 Scrolling through the customers

Above you can see the scrolled view of the list.

The solution in *Demo 09 CustomerManager List* implements the display of the customer list as we have seen above. If you run this you will see that scroll bars are automatically displayed, and that the screen can be flicked and squeezed in a most pleasing way. All this behaviour is provided for you by the ScrollViewer control.

Using the ListBox to display lists of items

We have seen that it is perfectly OK for an application to create display elements when it runs. Display elements can be described inside the XAML file, in which case they are made automatically when our program starts; or a program can create new element instances and add them to controls on the screen as it runs. We have just used this ability to display a list of customers but our program had to do all the hard work. It had to create all the display objects and then add them to the `StackPanel` where they were to be displayed.

Last time we had this problem we found we could use Data Binding to make life a lot easier. Data binding gets the display system to do all the hard work for us. We just have to bind the content of a display element to the data we want to display and it just works. Up until now we have just connected individual elements to single values (for example the result from the `AddingMachine` program). Now we are going to expand this to bind the contents of a customer object to a list.

However, this is a bit more complicated than the binding we did last time. The good news is that is actually very easy, and really, really powerful once you get the hang of it. So, let's do that.

Data Binding Single Items

In the adding machine program we used data binding to read numbers from the user (the content of the `TextBox` elements were bound to the value which needed to be added together) and we also used data binding to show the result of the calculation (the `AnswerValue` was bound to the `resultTextBlock`). We did all this in the XAML that described the screen display.

Above you can see the binding that we used in the adding machine to read the first number. The text in the `resultTextBlock` display element was bound to the `AnswerValue` property in the `Adder` class. When the class worked out a new answer, it was automatically displayed by data binding.

Creating a Data Template

We'd like to bind a `Customer` to a display element, but we can't just bind one directly because the value of a customer includes a bunch of different components. A `Customer` value contains a name, an address and a customer number. What we'd really like to do is design a template which sets out how the customer is to be displayed. This would let us choose which parts of a customer we want to display on the screen (perhaps we want to omit the customer number) and how they are formatted. Perhaps something like this:

```
<DataTemplate>
    <StackPanel>
        <TextBlock Text="{Binding Name}"/>
        <TextBlock Text="{Binding Address}"/>
    </StackPanel>
</DataTemplate>
```

This is a data template. It is a lump of XAML that describes how some data is to be displayed. We are indicating that we want to display the data in the form of a `StackPanel` that contains two items, the name and the address. It should result in a display a bit like this, with the two text strings stacked above each other:

```
Rob Miles
Rob Miles's House
```

Note that we are only displaying the name and address of a `Customer`. There is no data binding for the Customer ID value and so it will not be shown.

Using a DataTemplate in a list

We use a data template to indicate how to display properties of an object. We are going to use a template to display each of the customers in the list. We do this by placing this `DataTemplate` inside the `ItemTemplate` for the `ListBox` that will display our customers:

```
<ListBox Name="customerList">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="{Binding Name}"/>
                <TextBlock Text="{Binding Address}"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

OK. Time to step back little. The `ListBox` display element called `customerList` is going to display a list of our customers. To do this it needs to know how to lay out the display of a single `Customer`. A `ListBox` uses an `ItemTemplate` element to get the design of a list item, and this case it will be using a `DataTemplate` to do this.

So, once we have set up this XAML the `ListBox` just needs to be told the collection of data it is to display:

```
customers = Customers.MakeTestCustomers();
customerList.ItemsSource = customers.CustomerList;
```

This code creates a list of test customers and then sets the `ItemsSource` of the `customerList` to the list of customers. The `ListBox` will display each object in the collection as an element in the list.

Figure 4-31A customer list

Our program doesn't have to do anything other than just set the `ItemsSource` property of the `ListBox`. Everything else is done automatically. A `ListBox` even has scroll bars provided automatically, so if the list is too large for the screen the user can scroll up and down it.

From this example you should see how easy it is to display collections of data. You just have to design the template and then use data binding to get your data displayed.

Adding style to a display

The display above works OK, but it is not very interesting. It is also hard to see which are the customer names and which are the customer addresses. We could decide on different font sizes and styles for the name and address but we can also make use of some built in styles provided as resources for Windows Phone applications.

These styles have the advantage that they will work in different colour schemes and light and dark backgrounds. We add the style information to the TextBox items in the DataTemplate:

```
<DataTemplate>
    <StackPanel>
        <TextBlock Text="{Binding Name}"
                Style="{StaticResource PhoneTextExtraLargeStyle}"/>
        <TextBlock Text="{Binding Address}"
                Style="{StaticResource PhoneTextSubtleStyle}"/>
    </StackPanel>
</DataTemplate>
```

We are using `ExtraLargeStyle` for the customer name and `SubtleStyle` for the address. This has the result of making the display look much better.

Figure 4-32 A stylish customer list

If you want to, you can add lines and colours to make the display even nicer. They all go into the template and are drawn for each item in the list. If the source object contains images these can be added to the template too.

Selecting items in a ListBox

We can now display a list of items really easily. The next thing we want to do is make a proper data editing application. You will have seen this kind of application before. When the user selects one of the items in the list we want to open up a screen that allows this item to be edited:

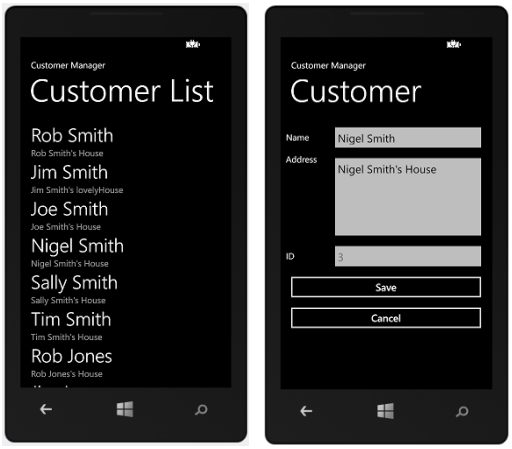


Figure 4-33 Customer List and edit item

The screens above show how this will work. The user scrolls up and down to find the customer they want to work with. They then select the required customer. The application then displays an edit screen for that customer. We can us this pattern any time we have a list of things that the user can pick one from.

It is very easy to get a `ListBox` to produce an event when an item is selected:

```
<ListBox Name="customerList"
         SelectionChanged="customerList_SelectionChanged">
```

This is where the `customerList` is declared in the XAML for the main page. The event hander must then be created inside the `MainPage.xaml.cs` source file.

```
private void customerList_SelectionChanged(object sender,
                                    SelectionChangedEventArgs e)
{
    // when we get here the user has selected a customer
    Customer selectedCustomer =
                    customerList.SelectedItem as Customer;

    MessageBox.Show(selectedCustomer.Name + " is selected");
}
```

The `customerList` provides a property called `SelectedItem` which is a reference to the object that has been selected. The method must convert this to a reference to a customer and then the program can display a message box containing this name.

The solution in *Demo 10 CustomerManager Binding List* implements the display of the customer list as we have seen above. MainPage.xaml contains the ListDisplay element and the template, and the program uses data binding to display the result. If we select an item in the list the program displays a MessageBox with the name of the item selected.

Programmer's Point: Selecting items in lists is easy

This pattern is very useful if you want to display a list of anything and have the user choose which one they want to work on.

The next thing we want to do for our customer manager application is take the user to a new page in our application that allows them to interact with the settings for this customer. To do this we need to know how to perform page navigation in Windows Phone applications.

4.7 Pages and Navigation

At the moment all the programs we have written have run on a single page. However, many applications run over a number of pages. The Customer Management application that we are creating will need a different page that will be used to display the information about the customer we are working on.

Adding a New Page

It is easy to add a new page to an application. If we select `Project>Add New Item` from within Visual Studio we are presented with a menu of possible items that we might like to add.

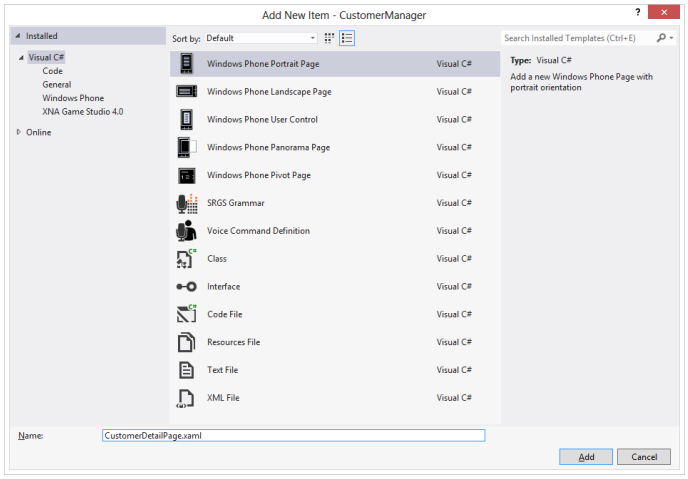


Figure 4-34 Adding a new page

If we select Windows Phone Portrait Page we get a new page added. Before adding a page it is sensible to set the name to something more meaningful than "Page1". We can change the name of the item later if we like by renaming it. An application can contain as many pages as we need.

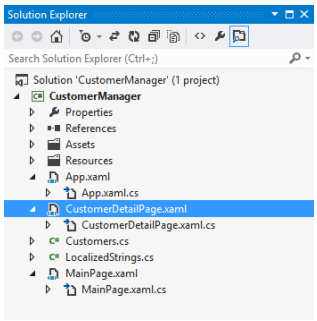


Figure 4-35 The new page in Solution Explorer

Here we can see that the project now contains a `CustomerDetailPage`. New pages themselves are edited in exactly the same way as `MainPage`. When the page is open we can drag elements onto it from the Visual Studio ToolBox, we can manipulate their properties and we can edit the XAML directly if we wish.

Navigation between Pages

The model for moving between pages has more in common with the internet than Windows Forms. If you are used to writing Windows Forms applications you will be familiar with methods such as "`Show`" and "`ShowDialog`" which are used to allow one form to cause the display of another.

Windows Phone XAML applications don't work like that. When you think about moving around a Windows Phone application you should think in terms of navigating from one page to another. Each page has an address expressed as a `uri` (Uniform Resource Indicator). The `NavigationService` object provides methods that perform the navigation for us.

```
// Navigate to the detail page
NavigationService.Navigate(new Uri("/CustomerDetailPage.xaml",
                    UriKind.RelativeOrAbsolute));
```

This code will cause the application to switch to a page called `CustomerDetailPage`. Note that the `Navigate` method is given a uri value which contains the name of the page and also the kind of Uri being used.

When we create the uri we also need to select what kind of uri it is, we are setting this uri to be `RelativeOrAbsolute`, even though this particular Uri is relative to the present one. We do this to ensure maximum flexibility when loading resources.

There is an issue here that you need to be aware of though. The Uri value is given as a string of text in your program. If you miss-type the string (for example put `"/CuystomerDeytailsPagerino.whamlxaml"`) the program will compile just fine, but then when it runs the navigation will fail with a run time error.

Using the Back button

Users will expect the Back button on Windows Phone to return to the previous page. This is exactly what it does. If the user has navigated to `CustomerDetailPage` as shown above, and then presses the Back button they will be returned to the page they came from. If the user presses Back when they are at the `MainPage` the application is ended.

Sometimes we want to override this behaviour. We might not want our user to leave the page if they have not saved the data they have entered. We can do this by adding an event handler to the `BackKeyPress` event for that page.

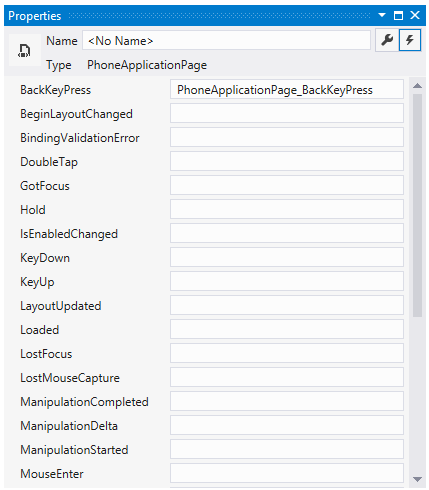


Figure 4-36 Page events, including Back button pressed

The event handler method can cancel the back behaviour if required.

```
private void PhoneApplicationPage_BackKeyPress(object sender,
                    System.ComponentModel.CancelEventArgs e)
{
    e.Cancel = true;
}
```

If the method above is bound to the `BackKeyPress` event the user will not be able to use the Back key to move away from the page.

Sometimes you want the user to confirm that they want to move away from the page. You can use the `MessageBox` class to do this:

```
private void PhoneApplicationPage_BackKeyPress(object sender,
                    System.ComponentModel.CancelEventArgs e)
{
    if (MessageBox.Show("Do you really want to exit the page?",
                        "Page Exit", MessageBoxButton.OKCancel)
        != MessageBoxResult.OK)
    {
        e.Cancel = true;
    }
}
```

This code displays a message box and asks the user if they really want to leave the page. If they don't want to leave the back button is cancelled. This is what the screen looks like when this code runs.

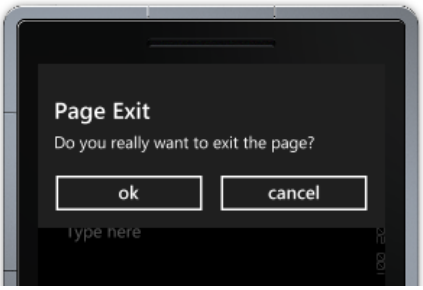


Figure 4-37 Message Box confirmation

You can use this form of `MessageBox` any time you want to put up a quick question for the user to answer.

Programmer's Point: Users expect your program to look after them

Adding confirmations like this is very important. Users will expect your program to ask them to confirm any action which could result in data loss.

Passing Data between Pages

Each page in an application is distinct and separate from any other. They do not share data. Sometimes you want to pass data from one page to another. If the data is a simple string of text the easiest way to do this is by adding it to the uri that is passed into the destination page. This technique is based on the way that web pages send queries to a web server.

As an example, we could consider our Customer Manager. When the user selects a customer from the list we would like to pass the name and address of the customer which has been selected into the edit page.

```
private void customerList_SelectionChanged(object sender,
                                           SelectionChangedEventArgs e)
{
    // Get the selected customer from the list
    Customer selectedCustomer = customerList.SelectedItem as Customer;

    // Build a navigation string with the customer information in it
    NavigationService.Navigate(new Uri("/CustomerDetailPage.xaml?" +
                                       "name=" + selectedCustomer.Name + "&" +
                                       "address=" + selectedCustomer.Address,
                                       UriKind.Relative));
}
```

This is the method that runs when the user selects a customer. The first thing it does is get the selected customer. Then it creates a Uri which contains the name and the address of the selected item. This is then used to navigate to the customer details page.

```
"/CustomerDetailPage.xaml?name=Sally Smith&address=Sally Smith's House"
```

This is the format of the Uri that would be created for Sally Smith. We can see that the query is the part after the `?` character in the Uri. This is then followed by two "name-value" pairs, separated by the `&` character. The first item is the name part of the query, and the second the address.

Now we need to understand how a receiving page can get hold of this information and display it. To do this we have to consider events that are raised when navigating pages.

Using Page Navigation Events

There are two really important events in the life of a page. One is fired when the page is navigated to (`OnNavigatedTo`) and the other is fired when the page is left (`OnNavigatingFrom`). To get control when these events fire we need to override these methods in our page.

When we create a new page the class that controls it is derived from a parent class called `PhoneApplicationPage`:

```
public partial class CustomerDetailPage: PhoneApplicationPage
{

    // CustomerDetailPage methods go here
    // we override the ones that we want to provide
    // our own behaviours for
}
```

This is the declaration of `CustomerDetailPage`. We add new methods to the page and we can override existing methods and add new behaviours to them. We want our application to get control when the user navigates to the page. This is the point where we want to load information from the uri and display it in a control on the page. Our replacement `OnNavigatedTo` method looks like this:

```
protected override void OnNavigatedTo(
                 System.Windows.Navigation.NavigationEventArgs e)
{
    string name, address;
    if (NavigationContext.QueryString.TryGetValue("name",
                                                  out name))
        nameTextBlock.Text = name;
    if (NavigationContext.QueryString.TryGetValue("address",
                                                  out address))
        addressTextBlock.Text = address;
}
```

The method uses the `QueryString` in the `NavigationContext` object and tries to get the data values out of that string. It asks for each value by name and, if a value is recovered, displays it in a text block. Each time the user returns to this page the method runs and puts the latest content onto the page. We can use the `OnNavigatedTo` event any time we want to set up a page when the user moves to it. We can pass as many named items into the destination page as we like.

One thing to remember is that we must make sure that the names match up in both pages. The compiler will not detect mistakes such as spelling the word `"address"` as `"adddresss"` in the above code. If we did this the program would run correctly, but the address information will not be displayed properly.

We can us this code to create a two page application that lets the user select a customer and see the details page for this customer. Unfortunately there is a tiny problem with page navigation if we do this. There is one behaviour that will be confusing:

1. Select an item from the list
2. Navigate to the customer details page.
3. Press the back button to return to the list
4. Select the same item on the list.

If the user does this they will notice that they are not returned to the same customer. They have to click a different customer and then return to the original one to select it. The reason for this is that the program is bound to the `SelectionChanged` event. As far as it is concerned, the selection has not changed when the user re-selects the same one. We can fix this by clearing the selection whenever this page is navigated to:

```
protected override void OnNavigatedTo(
                System.Windows.Navigation.NavigationEventArgs e)
{
    customerList.SelectedItem = null;
}
```

Now, if the user selects the same item again it will count as a valid selection event. Of course, setting the selection to null is actually a selection event and so if our program tries to use this property it will fail with a null exception. This is easy to fix; the event handler just has to make sure that an element in the list is selected before it tries to use it.

```
private void customerList_SelectionChanged(object sender,
                                    SelectionChangedEventArgs e)
{
    // Abandon if nothing selected
    if (customerList.SelectedItem == null) return;

    // Rest of method here
}
```

The above version of the method returns if there is no selected item.

The solution in *Demo 11 CustomerManager Query String* is a two page Windows Phone application. You can select a customer and see this customer displayed in the customer details page.

Programmer's Point: The user will have opinions on the user interface

At the beginning of a project a customer will probably say things like "Oh, just do it how you think best". Then, when you show them the end result of all your best thoughts they will suddenly reveal all kinds of strong opinions on how it should work. This often means that you end up re-writing large chunks of code. A better way is to work though each user interface with the user, pointing out issues like the one we have just discussed and making it very clear that once they have signed off the design it will be very expensive to change it. Then you just have to make the user interface once.

Sharing Objects between Pages

Passing strings of text between pages is useful, but you often need to share larger items which contain structured data. Often your user will be working with a "thing" of some kind; perhaps a document or a game, and the different pages will provide different views of that object. In our Customer Manager application we would really like to pass a `Customer` instance into the display page, so that it can work on the content of that customer. At the moment the display page is just being given the values of the items in the selected customer which are being passed as strings.

What you really want is an object which is common to all the pages and can be accessed by any of them. You could put your shared data into the object and they could all make use of it. Turns out that this is very easy to do, because all Windows Phone XAML programs have such a thing as part of the way they work. It is called the `App.xaml` page.

The App.xaml page

When we create a Windows Phone application we get a `MainPage.xaml` and an `App.xaml` page.

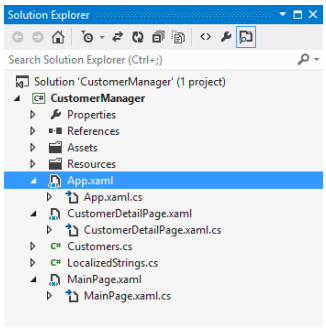


Figure 4-38 The App.xaml page

We can regard `App.xaml` as the container for the phone application. It provides the context in which the pages are displayed. It doesn't ever get drawn on the screen, but it is the starting point for the application when it is loaded. The `App.xaml.cs` file contains the methods that start an application running. It also contains handler methods for application lifecycle events which we will be using later.

We can edit the `App.xaml.cs` file and add our own code and data members. If we want to store any global data for use by all the pages in our application we can declare it here.

```
public partial class App : Application
{
    // To be used from all pages in the application
    public Customer ActiveCustomer;
}
```

Here we have added an `ActiveCustomer` value that we want to be able to use in all the pages. It will hold a reference to the customer that is currently being edited by the user. The `Current` member of the `Application` class is a reference that can refer to objects of type `Application`. When the application starts it is set to refer to the currently executing application instance. A program can use this to find the current application object.

```
private void customerList_SelectionChanged(object sender,
                              SelectionChangedEventArgs e)
{
    // quit if there is nothing selected
    if (customerList.SelectedItem == null) return;

    // Get the selected customer from the list
    Customer selectedCustomer =
            customerList.SelectedItem as Customer;

    // Get the parent application that contains the active customer
    App thisApp = Application.Current as App;

    // Set the active customer
    thisApp.ActiveCustomer = selectedCustomer;

    // Head to the details page
    NavigationService.Navigate(new Uri("/CustomerDetailPage.xaml",
                                       UriKind.Relative));
}
```

When a customer is selected the event handler now puts the customer reference into the App page and then navigates to the customer details page. The `App.xaml.cs` class defines a child class `App` that extends the `Application` parent. To get hold of objects defined in the `App` class we have to make a reference of type `App` and then make this refer to the current application. Above you can see how the as operator is used to convert the reference to one with the correct type. The destination page can use the same technique to find the current application and read back the active customer.

```
protected override void OnNavigatedTo(
                System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

    // Get the parent application that contains the active customer
    App thisApp = Application.Current as App;

    // Set the data context for the display grid to the active
    // customer
    customerDisplayGrid.DataContext = thisApp.ActiveCustomer;
}
```

This is the code that runs inside the `customerDetailsPage` when this page is navigated to. It gets the `ActiveCustomer` out of the application and sets the data context on the `customerDisplayGrid` to refer to the selected customer. We have already used data binding to allow us to directly connect all the customers to the customer selection page; here we are using it to display the values on a page.

Data Design

Since there may be other pages in the application that want to use the customer list it makes sense to put this list into the `App.xaml.cs` class as well.

```
// To be used from all pages in the application
public Customer ActiveCustomer;
public Customers AllCustomers = null;
```

The solution in *Demo 12 CustomerManager Shared Object* is a two page Windows Phone application that uses a shared variable to allow a user to select one of the items from the list and view this on the details page. It also uses a Grid control to layout the customer display. This is very useful if you want to align objects on a page.

4.8 Using ViewModel classes

At the moment our program is very good for displaying the customer, but it doesn't actually allow us to edit their details. The user can change the contents of the textboxes on the customer display page but the changes will not be reflected in the data in the program.

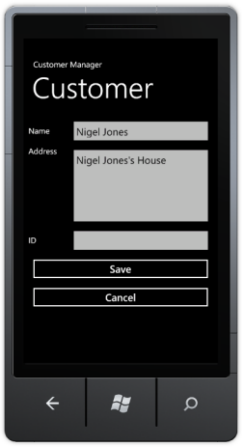


Figure 4-39 Editing a customer

What we really want to happen is that when the user updates the Address field, perhaps to "Nigel Jones's New House", this is automatically put back into the customer that is presently selected. We can do this with Data Binding; it is how we got the numbers out of the text boxes in the adding machine that we created earlier.

The first thing we do is tell that the binding of the text in the `TextBox` in the customer details page is two-way. This means that changes to the data will be displayed on the screen, and that changes in the `TextBox` will update the property in the class that is bound to it.

```
<TextBox Name="nameTextBox" Text="{Binding Name,Mode=TwoWay}"/>
```

Next we could add code to the `Customer` class to manage the binding events. This code would make sure that when we change the name the customer the text on the screen would change, and if the content of the `TextBox` changes the name property of the customer changes automatically. We did this in the Adding Machine with the `Adder.cs` class, and it seemed to work fine.

But first I think we need to stop and consider what we are doing.

Design with a ViewModel Class

We could add the data binding behaviours to the `Customer` class and from a programming point of view this would work fine. However, there are some reasons why it would not be a good idea from a design point of view.

Division of Purpose

There is a rule in software design that a class should usually do one thing; and one thing only. The job of the `Customer` class should be to hold customer data, not take part in an editing process. When we store or transfer customer data around we do not want to be worrying about methods and behaviours which are concerned with editing. If we transfer our customer onto another device for storage (and we will look at this later) any property binding behaviour will just be a waste of space, since it will not be used in this context.

Providing Validation and Data Conversion

Ideally something should make sure that what is entered into the form conforms to our business rules. These are things like "the name must contain only letters and spaces". We could put these rules into the `Customer` class, but it would be much easier if we could directly connect them to the data input process, since this is the point at which they are going to be used. It might also be useful to be able to perform conversion of some input values, for example dates and times, from the format that they are entered and displayed to the ones used for storage.

Providing an Undo Behaviour

This is a user interface issue, but it is also very important. If you look back at the edit screen in Figure 4.38 you will notice that there is a big `Save` button. The idea is that the user will press `Save` when they have completed editing, and this is the point that the changes they have made will be committed to the data. However, there is also a large `Cancel` button which would close the screen down and effectively cancel any changes.

If we are using direct data binding this would be a problem, as our data object would have already been changed to reflect what the user had entered. This means that we would have to make a copy of the data on entry to the form and then put it back in the object. Since the reason we are using data binding is to stop us having to write this kind of code, it seems a pity to have to start writing it all over again, just because the user wants to be able to change their mind.

Creating a ViewModel Class

We can address these problems by creating a *ViewModel* class. This has the job of representing a view of the data that we wish to present. It takes care of the data presentation and binding to the form components and can also perform any data validation as required. We create the ViewModel class to represent the activities we are going to perform in a particular user interface. We can think of it is a kind of "glue" that links the user interface and the actual data objects. It contains the properties that we wish to work with in the user interface. I've written one below called `CustomerView`.

```
public class CustomerView : INotifyPropertyChanged
{
    private string name;

    public string Name
    {
        get {
            return name;
        }
        set {
            name = value;

            if (PropertyChanged != null)
            {
                PropertyChanged(this,
                        new PropertyChangedEventArgs("name"));
            }
        }
    }

    // Address information here

    private int id;

    public int ID
    {
        get
        {
            return id;
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
```

```
    public void Load(Customer cust)
    {
        Name = cust.Name;
        Address = cust.Address;
        id = cust.ID;
    }

    public void Save(Customer cust)
    {
        cust.Name = Name;
        cust.Address = Address;
    }
}
```

This looks a lot like the Adder.cs class that we created earlier. It creates events when properties are changed by our program, and the controls can set new values in the class when they are changed by the user. Note that the ID property does not have a set behaviour. This is because the user is not allowed to change the ID of a customer. This is set when the customer is created and is unique for that customer.

Moving Data in and out of the Viewmodel

The `CustomerView` class also contains `Load` and `Save` methods. These are how the program sets the contents of the view class to the data being worked on. As an example, consider what happens when we start editing a customer:

```
// Viewmodel for the details page
CustomerView view = new CustomerView();

protected override void OnNavigatedTo(
                    System.Windows.Navigation.NavigationEventArgs e)
{
    // Get the parent application that contains the active customer
    App thisApp = Application.Current as App;

    // Load the active customer into the viewmodel
    view.Load(thisApp.ActiveCustomer);

    // Set the data context for the display to the viewmodel
    customerDisplayGrid.DataContext = view;
}
```

When the program moves to the customer details page the `OnNavigatedTo` method is called to set up the page. This method finds the currently active customer (this is the one that will have been selected) and then loads this into the `CustomerView` being used on this page.

When the user presses the `Save` button the program must take the content of the viewmodel and put it back into the customer being edited:

```
private void saveButton_Click(object sender, RoutedEventArgs e)
{
    // Get the parent application that contains the active customer
    App thisApp = Application.Current as App;

    // Copy the data from the viewmodel into the active customer
    view.Save(thisApp.ActiveCustomer);

    // Go back to the previous page
    NavigationService.GoBack();
}
```

ViewModels and Testing

The ViewModel class looks nice, but you might be wondering why we bother with it. It looks as if we can get a very similar behaviour just by putting the members of the Customer class directly into the `TextBox` elements on the page and then reading them back if the user presses the Save button.

While this is true, it overlooks one important aspect of program development, and that is testing. If the customer data editor interacts directly with the display elements it is very hard to automatically test whether it works or not. Test code would have to enter things into textboxes and look at the content of other textboxes to see if the application was doing what it was supposed to do. This becomes even more difficult if the user interface has to perform validation and data translation.

However, with a ViewModel class doing this work it is much easier to test. A test environment can simply connect to the same elements in the ViewModel as the user interface and inject test messages into it. There is no need to test by entering text onto the screen because a test environment can simply simulate the data binding.

Page Navigation using the GoBack method

The `NavigationService` provides a `GoBack` method that takes the user back to the previous page. This is more efficient than using a URI to navigate because it does not recreate the page. If you navigate to a page using the `Navigate` method provided by the `NavigationService` class this will create a new page each time you visit it. This can slow your application down and also lead to the creation and destruction of lots of display elements that will need to be recovered by the Garbage Collector. Furthermore the user will be a bit annoyed if, having scrolled down the list to select the item the want to work on, they are then taken back to the top of the list after an edit. I hate it when a badly written program does this kind of thing to me, and I'm sure you do to.

However, using the `GoBack` method to take us back to the original page does cause us problems. The difficultly is that the display on the original page will not reflect the changes that the user has made to the data. Consider the following sequence:

1. User selects customer Rob Miles
2. User changes name to "The Great Rob Miles" (and quite right too)
3. User Presses the Save button to perform the changes
4. Upon return to the selection screen the name is still "Rob Miles".
5. Confusion results.

The problem is that the `ListBox` is unaware that the content of a customer record has changed. When the customer list was displayed it made a copy of the data onto the screen and it does not know that this needs to be updated. The actual data has been updated, but the view of it does not reflect this. To solve the problem we have to go even deeper into XAML and find out how to use an `ObservableCollection` to solve the problem

Observable Collections

We have seen that the best way to connect a data object to a page in the application is to put a ViewModel class in between the two. The ViewModel can be customised for the particular view required, and the data object can focus on just storing the data.

This is also true when considering collections of data. At the moment we are binding a list of customers directly to the `ListBox`. This is fine for just viewing the content of a list, but as we have seen, if any elements of the list change there we do not have a notification method that can be used to indicate when the display on the screen needs to be updated. What we need is another ViewModel class that can represent the list of customers and provide a way that the `ListBox` can be notified of changes to elements in it.

This is what the `ObservableCollection` was created for. It can be used to hold a collection of items and provides notification support so that if the content of the collection changes the `ListBox` can be bound to change events. We can create an `ObservableCollection` of customers very easily:

```
ObservableCollection<Customer> observableCustomers;
```

```
private void PhoneApplicationPage_Loaded(object sender,
                                         RoutedEventArgs e)
{
    // Get the parent application that contains the customers list
    App thisApp = Application.Current as App;

    // Make an observable collection of these
    observableCustomers =
        new ObservableCollection<Customer>(
                        thisApp.ActiveCustomerList.CustomerList);

    // Set the list to display the observable collection
    customerList.ItemsSource = observableCustomers;
}
```

This is the code that runs when the main page of our Customer Manager starts. It creates a new list called `observableCustomers` from the customer list. It then sets the source of the `customerList` to this newly created list. Now, if the program makes changes to the list the display will update to reflect this.

Unfortunately the program still doesn't reflect changes we want. If the user updates a name in a customer this will not be reflected by the text on the display. This is because the `ObservableCollection` responds to the changes in the contents of the list, not changes to the data in an element in the list.

Actually, what you get is worse than just the display not updating correctly. What will happen is that if the `ListBox` ever needs to redraw that element it will go back to the original data and display it. This means that if the user scrolls up and down the list for a while there is a good chance that the correct data will appear eventually, as the `ListBox` will only create display objects that are visible, and may discard ones that have been moved off the screen. This can lead to some very confusing effects where the program seems to catch up after a while.

To get the view to update we have to do something to the content of the list to force the update to take place. One way to force an update is to remove something from the list and put it back again. Changes to the content of an observable collection generate events that are picked up by the `ListBox` that is displaying the list.

```
protected override void OnNavigatedTo(
                    System.Windows.Navigation.NavigationEventArgs e)
{
    // Get the parent application that contains the customer list
    App thisApp = Application.Current as App;

    if (thisApp.ActiveCustomer != null)
    {
        // Find the customer in the list
        int pos = observableCustomers.IndexOf(
                                        thisApp.ActiveCustomer);
        // Remove it
        observableCustomers.RemoveAt(pos);
        // Put it back again - to force a change
        observableCustomers.Insert(pos, thisApp.ActiveCustomer);
    }
}
```

This is the code that runs when the user navigates back to the main page. If there is an active customer (i.e. one has been selected for edit) the method removes this customer from the list and then puts it back in the same place. This forces the redraw of the item and the user will now see changes reflected immediately. Even better, because we are just updating the one item, the display will update really quickly. This is particularly important if we have a list which contains many thousands of customers.

Saving Data

When we used a `ViewModel` class to edit a customer we had Load and Save methods that put the customer into the `CustomerView` class and then recovered the updated values from it. In the case of the `ObservableCollection` we have to do something similar. Fortunately there are a number of ways that we can ask the `ObservableCollection` to give back the data it is displaying.

```
thisApp.ActiveCustomerList.CustomerList =
   observableCustomers.ToList<Customer>();
```

The `ToList` method extracts the list of customers and returns it.

Observable Collections and Efficiency

Programmer's Point: Observable Collections don't slow things down

You might be thinking that adding an ObservableCollection makes the program run more slowly and use up more memory. This is actually not the case. The collection is another set of references to the same customer objects that are in the customer store. This means that the only extra memory that is required is for the observable collection itself, which is not large.

If your program only needs to view the content of a list then you do not need to use the `ObservableCollection`. It is only when the contents of the list need to change because of updates to the data in the list that you need to add this.

The solution in *Demo 13 Complete CustomerManager* is a two page Windows Phone application that allows the user to select and edit items in the list of customers. Changes to an item are updated in the list view and the user is returned to the original position at the end of each edit.

What We Have Learned

1. Programs can manipulate the properties of elements to display information to the user of a program. This includes the position and colour of items on the display.
2. Some element properties are best set by editing the XAML directly. The XAML information for an element is structured in terms of properties which can be nested inside each other. The TextBox has a set of properties that give the initial settings for the keyboard to be used to enter data. These can be used to cause a numeric keyboard to be displayed instead of a text one.
3. Windows Phone can display messageboxes to the user. These can display multi-line messages and can also be used to confirm or cancel user actions.
4. Assets such as images can be added to a Windows Phone application as content or resources. An item of content is copied into the application directory as a separate file and can be used from there by the program. A resource item is embedded into the assembly file for an application. Content items do not slow the loading of assemblies but may be slower to load when the program runs. Resource items are more readily available to a program but they increase the size of the program assembly.
5. Display elements can generate events in response to user actions. One such action is the TextChanged event produced by the TextBox.
6. XAML provides data binding support where the properties of an object in the program can be connected to those of a display element. The binding can be "one way", where the display element is used to output the value of a program item or "two way" where changes to the item on the page result in an update of the bound property in the class.
7. It is possible to bind a collection of items to a ListBox element to show a list of items. A data template is used to express how the individual data values in each item are to be displayed.
8. Applications can be made up of multiple pages. The navigation between pages is performed by a navigation helper class which is given the uri of the page to be navigated to. Simple items of text can be passed between pages by placing them in a query string attached to the uri.
9. Pages can receive events when they are navigated to and from. The OnNavigatedFrom event provides cancellation option so that a user can be asked to confirm the navigation.
10. Larger data objects can be shared between pages in an application by the use of the App class which is part of a Windows Phone application. Any page in an application can obtain a reference to the application object that it is part of.
11. Programmers can create ViewModel classes that are set to the data being edited and bound to the XAML controls on a page. These act as "glue" between the actual data in the program and the particular presentational and editing needs of the user interface being developed.

5 Isolated Storage on Windows Phone

Each application or game has its own particular data storage needs. The storage can be as simple as a name – value pair (for example HighScore=100). On the other hand it you might be storing an entire product database along with all your customers. Or you might be storing text documents or pictures taken with the phone camera.

In this section we are going to explore the ways in which C# applications on the device can connect to and use data services provided by the phone.

5.1 Storing Data on Windows Phone

A proper application will need to store data which it can use each time it runs. An application will need to hold settings and information the user is working on and a game will want to store information about the progress of a player through the game along with high score and player settings. Windows Phone programs can use "isolated storage" to hold this kind of information. The storage is called "isolated" because it is not possible for one application to access data stored by another. This is different from a Windows PC where any program can access any file on the system.

The Isolated Storage area can hold very large amounts of data, up to the limit of the storage available in the phone itself. A Windows Phone will have at least 8G of built in storage which is shared between media (stored music, pictures and videos) and all the applications on the device.

When an application is removed from the phone all the isolated storage assigned to it is deleted. When an application is upgraded (i.e. we release a new version of our program or game) the isolated storage retains the original contents. If the data there must be updated for the new version our application must do this.

Using the Isolated Storage File System

You can use the isolated storage in the same way as you use a file system. The only difference is the way that a program makes a connection to the storage itself. However, once a program has a connection to the file system it can use it as it would any other, creating streams to transfer data to files and even using folders to create a structured filestore. A program can store very large amounts of data in this way.

As an example we could create a simple application which gives the user a simple jotter where they store simple messages. This version of the program uses a single file but could easily be extended to provide a note storage system.

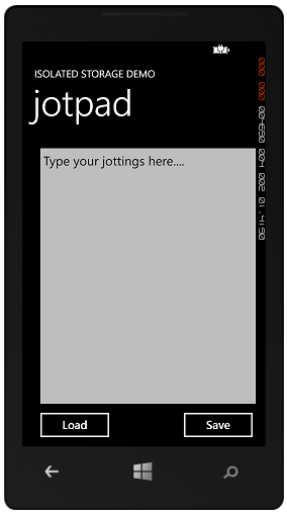


Figure 5-1 Jotpad in action

The user can type in jottings which are saved when the Save button is pressed. If the Load button is pressed the jottings are loaded from storage.

```
private void saveButton_Click(object sender, RoutedEventArgs e)
{
    saveText("jot.txt", jotTextBox.Text);
}
```

This is the code for the Save button. It takes the text out of the textbox and passes it into a call of the `saveText` method along with the name of the file to store it in.

```
private void saveText(string filename, string text)
{
    using (IsolatedStorageFile isf =
                    IsolatedStorageFile.
                        GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream rawStream =
                            isf.CreateFile(filename))
        {
            StreamWriter writer = new StreamWriter(rawStream);
            writer.Write(text);
            writer.Close();
        }
    }
}
```

The `saveText` method creates a stream connected to the specified file in isolated storage and then writes the text out to it. If you have used streams in C# programs already you will be pleased to find that these work in exactly the same way. In this case the method creates a `StreamWriter` and then just sends the text out to that stream.

The read button uses a `loadText` method to do the reverse of this operation.

```
private void loadButton_Click(object sender, RoutedEventArgs e)
{
    string text;

    if ( loadText("jot.txt", out text ) )
    {
        jotTextBox.Text = text;
    }
    else
    {
        jotTextBox.Text = "Type your jottings here....";
    }
}
```

The `loadText` method tries to open a file with the name it has been given. It then tries to read a string from that file. If any part of the read operation fails the `loadText` method returns `false` and `jotTextBox` is set to a starting message.

If the file can be read correctly the `loadText` method returns `true` and sets the output parameter to the contents of the file. This is then used to set the text in `jotTextBox`.

Note that we are using an `out` parameter to allow the `loadText` method to return the string that was read as well as whether it worked or not.

```
private bool loadText(string filename, out string result)
{
    result = "";
    using (IsolatedStorageFile isf =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (isf.FileExists(filename))
        {
            try
            {
                using (IsolatedStorageFileStream rawStream =
                    isf.OpenFile(filename,
                                 System.IO.FileMode.Open))
                {
                    StreamReader reader =
                                new StreamReader(rawStream);
                    result = reader.ReadToEnd();
                    reader.Close();
                }
            }
            catch
            {
                return false;
            }
        }
        else
        {
            return false;
        }
    }
    return true;
}
```

The `loadText` method will return `false` if the input file cannot be found or the read process fails. Note that it uses the `ReadToEnd` method to read from the file so that the program can read multi-line jottings.

The solution in *Demo 1 Jotpad* contains a Windows Phone jotter pad that stores text in a file and loads it back. If you stop the program and run it again from the Application list in the emulator you will notice that the stored data is recovered when you press Load. Note that you could expand the save and load methods to store even more data simply by adding extra write and read statements and further parameters. Alternatively you could make the load and save methods work on an object rather than a number of individual items.

Using the Isolated Storage Settings Storage

Quite often the only thing a program needs to store is a simple settings value. In this case it seems a lot of effort to create files and streams just to store a simple value. To make it easy to store settings a Windows Phone program can use the Isolated Storage settings store. This works like a *Dictionary*, where you can store any number of name/value pairs in the isolated storage area.

An introduction to dictionaries

Dictionaries are a very useful collection mechanism provided as part of the .NET framework. A dictionary is made using two types. One is the type of the key, and the other is the type of the item being stored. For example, we might create a class called `Person` which holds data about, surprise surprise, a person:

```
class Person
{
    public string Name;
    public string Address;
    public string Phone;
}
```

This Person class above only contains three fields, but it could hold many more. We would like to be able to create a system where we can give the name of a Person and the system will then find the Person value with that name. A Dictionary will do this for us very easily:

```
Dictionary<string, Person> Personnel =
                            new Dictionary<string, Person>();
```

When we create a new Dictionary type we give it the type of the key (in this case a string because we are entering the name of the person) and the type of the item being stored (in this case a Person). We can now add things to the dictionary:

```
Person p1 = new Person { Name = "Rob", Address = "His House",
                         Phone = "1234"
                       };

Personnel.Add(p1.Name, p1);
```

This code makes a new Person instance and then stores it in the Personnel dictionary. Note that the code takes the name out of the person that has been created and uses that as the key. The program can now use a string as an indexer to find items in the dictionary:

```
Person findPerson = Personnel["Rob"];
```

The findPerson reference is set to refer to the Person instance with the name Rob. The dictionary does all the hard work of finding that particular entry. This makes it very easy to store large numbers of items and locate them later.

The dictionary will refuse to store an item with the same key as one already present. In other words if I try to add a second person with the name "Rob" the Add operation will throw an exception. A program will also get an exception if it asks the dictionary for a record that doesn't exist:

```
Person findPerson = Personnel["Jim"];
```

Fortunately there is also a mechanism for determining whether or not a given key is present in the dictionary:

```
if (Personnel.ContainsKey("Jim"))
{
    // If we get here the dictionary contains Jim
}
```

Dictionaries are very useful for storing name-value pairs. They remove the need to write code to search through collections to find things. We can use multiple dictionaries too, for example we could add a second dictionary to our Personnel system that would allow us to find a person from their phone number.

Dictionaries and Isolated Storage

The IsolatedStorageSettings class provides us with a dictionary based storage system for settings that we can use to store setting values. The settings dictionary stores a collection of objects using a string as the key.

Saving text in the isolated storage settings

We could convert our jotter to use the settings class as follows:

```
private void saveText(string filename, string text)
{
    IsolatedStorageSettings isolatedStore =
                    IsolatedStorageSettings.ApplicationSettings;
    isolatedStore.Remove(filename);
    isolatedStore.Add(filename, text);
    isolatedStore.Save();
}
```

This version of the saveText method uses the filename as the key. It removes an existing key with the given filename and then adds the supplied text as a new entry. The Remove method can be called to remove an item from a dictionary. It is given the key to the item to be removed. If the key is not present in the dictionary the Remove method returns false, but we can ignore this. Once we have made our changes to the store we need to call the Save method to persist these changes.

Loading text from the isolated storage settings

The `loadText` method will fetch the value from the settings dictionary:

```
private bool loadText(string filename, out string result)
{
    IsolatedStorageSettings isolatedStore =
              IsolatedStorageSettings.ApplicationSettings;
    result = "";
    try
    {
        result = (string)isolatedStore[filename];
    }
    catch
    {
        return false;
    }
    return true;
}
```

The `loadText` method fetches the requested item from the settings storage. It is made slightly more complicated by the fact that, unlike the `Dictionary` class, the `IsolatedStorageSettings` class does not provide a `ContainsKey` method that we can use to see if a given item is present. The above method simply catches the exception that is thrown when an item cannot be found and returns false to indicate that the item is not present. Note that it must cast to string the value that is loaded from the settings dictionary. This is because the dictionary holds objects (so that we can put any type into it).

We now have two ways to persist data on a Windows Phone device. We can store large amounts of data in a filestore that we can create and structure how we like. Alternatively we can store individual data items by name in a settings dictionary. The isolated storage mechanism can be used by all Windows Phone programs.

The solution in *Demo 02 Settings JotPad* contains a Windows Phone jotter pad that stores text in the IsolatedSettings dictionary.

The Isolated Storage Explorer

When we are developing an application it is useful to be able to see the files that have been stored there. The Windows Phone Developer Toolkit contains a tool that lets you do just that. It is a program that is installed with the Windows Phone developer toolkit. On my system I can find it on the path below:

```
C:\Program Files (x86)\Microsoft SDKs\Windows
Phone\v8.0\Tools\IsolatedStorageExplorerTool
```

The Isolated Storage Explorer is given the Product GUID (Globally Unique Identifier) of the application. This tells it which area of Isolated Storage to read. We can find the product GUID in the `WMAppManifest.xml` file for the application:

```
<App xmlns="" ProductID="{3363ac33-4f45-4b21-b932-fa2084b6deb0}"
Title="JotPad" RuntimeType="Silverlight" Version="1.0.0.0" Genre="apps.normal"
Author="JotPad author"
Description="Sample description" Publisher="JotPad">
```

To read the contents of an isolated store we issue the `ISETool` command with appropriate parameters.

```
ISETool ts xd 3363ac33-4f45-4b21-b932-fa2084b6deb0 c:\isoStore
```

This command takes a snapshot (`ts`) of the emulator (`xd`) isolated storage for the Jotpad application above and places it in the folder `c:\isoStore`.

There are lots of other things you can do with the explorer. The full description of the command is as follows:

```
ISETool.exe <cmd[:param]> <target-device[:param]> <product-id> [<desktop-path>]


  <cmd[:param]> - Specifies the command to be executed (one of the following)
    ts                - (takesnapshot) to download the contents of isolated
```

```
                      store from <target-device> to desktop
  rs                - (restoresnapshot) to upload the contents of isolated
                      store from desktop to <target-device>
  dir[:device-path] - lists the contents of the device folder or root
                      if not mentioned
  EnumerateDevices  - lists the valid device targets along with their
                      device indices.

<target-device[:param]> - Specifies the target device (one of the following)
  xd                - default emulator
  de                - Windows Phone device connected to the desktop
  deviceindex:n     - device listed at index n. To get the list of devices
                      use the following command
                       "ISETool EnumerateDevices"

<product-id> - Specifies the GUID of the product. This is located in
               WMAppManifest.xml file of the project

<desktop-path> - desktop path for download and upload
```

The built-in help for the command has some examples of how the program is used.

5.2 Copying Files into Isolated Storage

It is sometimes necessary to copy files from your program into isolated storage. For example, if you want to use a background thread to play music the music files must be stored in the isolated storage region.

```
private void CopyToIsolatedStorage()
{
    using (IsolatedStorageFile storage =
IsolatedStorageFile.GetUserStoreForApplication())
    {
        string[] files = new string[] { "everythingSound.mp3" };

        foreach (var _fileName in files)
        {
            if (!storage.FileExists(_fileName))
            {
                string _filePath = "Audio/" + _fileName;
                StreamResourceInfo resource =
                    Application.GetResourceStream(new Uri(_filePath,
                                                  UriKind.Relative));

                using (IsolatedStorageFileStream file =
                                        storage.CreateFile(_fileName))
                {
                    int chunkSize = 4096;
                    byte[] bytes = new byte[chunkSize];
                    int byteCount;

                    while ((byteCount = r
                                  esource.Stream.Read(bytes, 0, chunkSize)) > 0)
                    {
                        file.Write(bytes, 0, byteCount);
                    }
                }
            }
        }
    }
}
```

This method is given a list of files that are held as content in the application and then copies them into isolated storage. The version above puts them in a folder called Audio, although you can store them where you like.

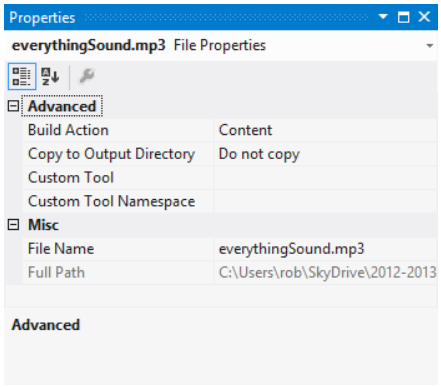


Figure 5-2 Setting the build action to Content

If you do this you must set the build action for the resource to Content, as shown above in Figure 5-2 .

What We Have Learned

1. A Windows Phone provides a system of isolated storage by which each application on the phone is able to store its own data on the device.
2. Programs can create folders and files in their isolated storage area and use standard input/output constructions based on streams to communicate with them.
3. An alternative method to store data is to make use of a dictionary based mechanism to store name-value pairs such as settings values (Background colour = blue).

4. The Isolated Storage Explorer tool allows programmers to view the contents of the isolated storage for an application under development.

6 Using Databases on Windows Phone

6.1 An Overview of Database Storage

A database is a collection of, well, data, which is organised and managed by a computer program which is often, and rather confusingly, called a database.

Another computer program can ask a database questions and the database will come back with the results. You don't think of a database as part of your program as such, it is the component in your solution that deals with data storage.

We could discover how a database works by taking a quick look at a very simple example. Consider how we would create a database to hold information for an internet shop. We have a large number of customers that will place orders for particular products.

If we hire a database designer they will charge us a huge amount of money and then they will come up with some designs for database tables that will hold the information in our system. Each of the tables will have a set of columns that hold one particular property of an item, and a row across the table will describe a single item. This is the design for the customer table:

Customer ID	Name	Address	Bank Details
123456	Rob	18 Pussycat Mews	Nut East Bank
654322	Jim	10 Motor Drive	Big Fall Bank
111111	Ethel	4 Funny Address	River Bank

This is a **Customers** table that our sales system will use. It contains the same information that was stored in the Customer Manager, with the addition of a string which gives bank details for that customer. Each row of the table will hold information about one customer. The table could be a lot larger than this; it might also hold the email address of the customer, their birthday and so on.

Product ID	Product Name	Supplier	Price
1001	Windows Phone 7	Microsoft	200
1002	Cheese grater	Cheese Industries	2
1003	Boat hook	John's Dockyard	20

This is the **Products** table. Each row of this table describes a single product that the store has in stock. This idea might be extended so that rather than the name of the supplier the system uses a Supplier ID which identifies a row in the Suppliers table.

Order ID	Customer ID	Product ID	Quantity	Order Date	Status
1	123456	1001	1	21/10/2010	Shipped
2	111111	1002	5	10/10/2010	Shipped
3	654322	1003	4	1/09/2010	On order

This is the **Orders** table. Each row of the table describes a particular order that has been placed. It gives the Product ID of the product that was bought, and the Customer ID of the customer that bought it.

By combining the tables you can work out that Rob has bought a Windows phone, Ethel has bought five Cheese graters and the four Boat hooks for Jim are on order.

This is a bit like detective work or puzzle solving. If you look in the Customer table you can find that the customer ID for Rob is 123456. You can then look through the order table and find that customer 123456 ordered something with an ID of

1001. You can then look through the product table and find that product 1001 is a Windows Phone (which is actually not that surprising).

Databases and Queries

You drive a database by asking it questions, or queries. We could build a query that would find all the orders that Rob has placed. The database system would search through the Orders table and return all the rows that have the Customer ID of 123456. We could then use this table to find out just what products had been purchased by searching through the Products table for each of the Product IDs in the orders that have been found. This would return another bunch of results that identify all the things I have bought. If I want to ask the database to give me all the orders from Rob I could create a query like this:

```
SELECT * FROM Orders WHERE CustomerID = "123456"
```

This would return a "mini-table" that just contained rows with the Customer ID of "123456", i.e. all the orders placed by Rob. The commands I'm using above are in a language called SQL or Structured Query Language. This was specifically invented for asking questions of databases.

Companies like Amazon do this all the time. It is how they manage their enormous stocks and huge number of customers. It is also how they create their "Amazon has recommendations for you" part of the site, but we will let that slide for now.

Connecting to a Database

Unfortunately object oriented programs do not work in terms of tables, rows and queries. They work in terms of objects that contain data properties and methods. When a program connects to a database we need a way of managing this transition from tables to objects.

Databases and Classes

We have already seen that we can represent data in classes. The `Customer` class in the Customer Manager application we have been working on holds information about each customer:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string BankDetails { get; set; }

    public Customer(int inID, string inName, string inBank,
                    string inAddress,)
    {
        Name = inName;
        Address = inAddress;
        BankDetails = inBank;
        ID = inID;
    }
}
```

To hold a large number of customers we created a class that holds a list of them.

```
List<Customer> Customers = new List<Customer>();
```

Whenever I add a new customer I add it to the `Customers` list and to find customers I use the `foreach` loop to work through the customers and locate the one I want. As an example, to find all the orders made by a customer I could do something like this:

```
public List<Order> FindCustomerOrders(int CustomerID)
{
    List<Order> result = new List<Order>();

    foreach ( Order order in Orders )
    {
        if (order.CustomerID == CustomerID)
        {
            result.Add(order);
        }
    }
    return result;
}
```

The method searches through all the orders and builds a new list containing only ones which have the required customer ID. It is a bit more work than the SQL query, but it has the same effect.

Using LINQ to connect Databases to Objects

People like using databases because it is easier to create a query than write the code to perform the search. However, they also like using objects because they are a great way to structure programs. To use databases and object oriented programs together the programmer would usually have to write lots of "glue" code that transferred data from SQL tables into objects and then back again when they are stored. This is a big problem in large developments where they may have many data tables and classes based on information in the tables.

Language Integrated Query, or LINQ, was the result of an effort to remove the need for all this glue. It is called "Language Integrated" because they actually had to change the design of the C# language to be able to make the feature work.

Windows Phone supports the creation of databases for our programs to use and all the database interaction is performed using LINQ. So now we are going to see how we can take our class design and use it to create a database. We are going to use SQL to store the customers in our Customer Manager. Then we are going to add support for product orders that they place. This will let our user hold a large number of customers, orders and products in a database on the phone.

The next thing we are going to do is start to design the database tables, but before we can do that we need to make LINQ work with this project.

The LINQ Libraries

The LINQ libraries are part of the Windows Phone libraries that are included when a solution is built. This means that you don't need to add them to your project. Note that this is a change from earlier versions of Windows Phone, where you had to add the `System.Data.Linq` libraries to the project.

We can however add some "using" directives to make it easier to access the classes in this library.

```
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.ComponentModel;
using System.Collections.ObjectModel;
```

We can now use all the classes from these libraries without having to use the fully qualified form of each name.

Creating a LINQ Table

Next we want to create some table designs that our database is going to manage. There will be three tables in our database.

- Customers
- Orders
- Products

We could start by considering how we are going to store information about a customer. From the original table we created a `Customer` class that describes the data a customer must contain. Now we want to use this class to create a design for a database table that will hold all our customers. We can do this by taking our original `Customer` class and modifying it so that LINQ can use the class design to create a table in the database from it. This is our `Customer` class design.

```
public class Customer
{
    public string Name {get; set;}
    public string Address { get; set; }
    string BankDetails { get; set; }
    public int CustomerID { get; set; }
}
```

This class holds three strings and an integer, which are all members of the class. They are implemented as properties, with `get` and `set` behaviours. The start of our `Customer` class for LINQ will look rather similar:

```
[Table]
public class Customer : INotifyPropertyChanged,
                        INotifyPropertyChanging
{
    // Customer table design goes here
}
```

The [Table] item is an *attribute*. Attributes are used to flag classes with information which can be picked up by programs that look at the *metadata* in the compiled code. Metadata, as I am sure you are all aware, is "data about data". In this case the data being added is the attribute, and the data it is about is the `Customer` class.

When a C# class is compiled the compiler adds lots of metadata to the output that is produced, including the precise version of the code, details of all the methods it contains and so on. Other programs can read this metadata, along with the class information.

The code above adds the [Table] attribute to the `Customer` class, which LINQ interprets as meaning "This class can be used as the basis of a data table". There is actually nothing much inside the [Table] attribute itself, it just serves as a tag.

The code above also states that `Customer` class implements the `INotifyPropertyChanged` and `INotifyPropertyChanging` interfaces. A class implements an interface when it contains all the methods that are specified in the interface. The two interfaces contain methods that will be used to tell LINQ when the content of the data in the class are being changed.

We have seen this situation before, where data bound to a display element needed to communicate changes so that the display updated automatically when the data was changed. This situation is very similar, except that a change to the data is going to trigger updates in the database.

Each interface contains a single event delegate. This is the delegate for `INotifyPropertyChanged`.

```
public event PropertyChangedEventHandler PropertyChanged;
```

The LINQ infrastructure will bind to the `PropertyChanged` event so that it can be told when a property value has changed. There is also a delegate to be used to tell LINQ when a value is changing. This event is fired before the change is executed:

```
public event PropertyChangingEventHandler PropertyChanging;
```

Our job is to make sure that the `Customer` class fires these events when data changes. In other words, if a programmer does something like this:

```
activeCustomer.Name = "Trevor";
```

- the code that updates the `Name` property should also tell LINQ that the data has changed. This means that the `Name` property could look like this:

```
private string nameValue;

public string Name
{
    get
    {
        return nameValue;
    }
    set
    {
        if (PropertyChanging != null)
        {
            PropertyChanging(this,
                  new PropertyChangingEventArgs("Name"));
        }
        nameValue = value;
        if (PropertyChanged != null)
        {
            PropertyChanged(this,
                  new PropertyChangedEventArgs("Name"));
        }
    }
}
```

The `Get` behaviour for the property just returns the value of the data member that holds the name.

The `Set` behaviour tests to see if anything is attached to the event handlers for the events and calls one before the property is changed, and the other after. The delegate calls are supplied with two parameters. The first is `this`, which is a reference to the currently active `Customer` instance (the one whose name is changing). The second is an event argument that contains the name of the property that is being changed. LINQ can use these to work out what has changed and perform an appropriate database update. As we saw when using notification events, if the name of the property is specified incorrectly (we say “name” or “nayme”) the update will not work correctly.

The change generates “before” and “after” events so that LINQ can manage the data changes more efficiently. If this seems like a lot of extra work, remember that the result of our efforts will be that when we update objects from our database we don’t have to worry about storing the effects of our changes. This will all happen automatically.

If our objects contain a lot of data fields it makes sense to simplify the changed event management by writing some methods that do the notification management for us:

```
private void NotifyPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this,
                    new PropertyChangedEventArgs(propertyName));
    }
}

private void NotifyPropertyChanging(string propertyName)
{
    if (PropertyChanging != null)
    {
        PropertyChanging(this,
                    new PropertyChangingEventArgs(propertyName));
    }
}
```

We can then simplify the name property to this behaviour:

```
private string nameValue;

public string Name
{
    get
    {
        return nameValue;
    }
    set
    {
        NotifyPropertyChanging("Name");
        nameValue = value;
        NotifyPropertyChanged("Name");
    }
}
```

If we make use of TwoWay data binding the result will be that once we have connected our data objects to the display elements they will update themselves automatically, which is very useful indeed.

There is one final thing we need to add to the `Name` property to allow it to be used by LINQ. We have to mark the property with the `[Column]` attribute, so that LINQ knows that it is to be used in the database:

```
[Column]
public string Name
{
    // Name property behaviour goes here.
}
```

The address and bank details items can be added to the `Customer` class in exactly the same way. Each of them will map onto another column in the database.

Creating a Primary Key

The CustomerID item is slightly different however. In our original Customer Manager it was an integer value that identified each customer. If the customer gets married and changes their name the customer ID is there to make sure we can still find them in the application. In database terms we are using the Customer ID as the *primary key* for this table. The primary key is used to uniquely identify particular customers in this table. We might have two customers with the name "John Smith", but we will **never** have two customers with the same ID. The ID works in exactly the same way as your Social Security number or bank account number.

When we create the ID column we have to tell LINQ that this is the primary key for the table. We can also ask LINQ to make sure that all values in this column are unique, and we can even ask that the values for this element be auto-generated. We do this by adding information to the Column attribute for the CustomerID property:

```
[Column(IsPrimaryKey = true, IsDbGenerated = true)]

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

When we add a new customer entry to the database we don't have to worry about the ID, as it will be generated for us. We can of course view the ID value, so that we can tell the next "John Smith" what their unique id is.

Note that we are **never** allowed to change the ID for a customer. This is a unique key which will be automatically created by the database for each customer in turn. In database terms this property is called the *primary key* for the customer record. The primary key of a customer will be used in our database to implement a relationship between the Customers table and the Orders table. Each row of the Orders database will contain a customer ID value that identifies the customer who placed that order.

When LINQ reads the metadata for this property it will use these settings to decide what kind of database column to create.

Creating a LINQ Data Context

Now that we have the design for a row in the table we can create the table itself. In our previous customer manager application we used a List to hold the customers. In LINQ we create a `DataContext` which will manage the connection to the database we are about to create. This will contain all the tables in the application, which at the moment just means the customers:

```
public class SalesDB : DataContext
{
    public Table<Customer> Customers;

    public Customers(string connection) : base(connection)
    {
    }
}
```

The `SalesDB` class extends the `DataContext` class, which is provided by LINQ as the basis of a database design. The constructor for this class simply calls the constructor for the parent class. The connection string gives location of the database we are going to use. One of the great things about database connection strings is that they can be used to connect to databases over a network, so that a program can work with a database on a distant server. We are not going to do this; we are going to connect the database to a file held in isolated storage on the phone.

The `DataContext` serves as a connection to a database. You can think of it as a bit like a stream that connects a program to a file. It exposes methods that we can use to manage the database contents.

The connection string describes how the program will connect to the database. In some applications this connection can be to a database server on a distant machine. In this case the database requests are sent to the remote system and the program works with the responses. In the case of the database on a Windows Phone this is a path to a file held in Isolated Storage.

Creating Sample Data

When we created our original Customer Manager program we created a method to create sample data. We are going to do exactly the same thing with our database version. This will populate the database with known data which we can then work with. Note that the method is now called `MakeTestDB`, because we will add further test setup for the other tables when we add them later. We can start with exactly the same sample information:

```
public static void MakeTestDB(string connection)
{
    string[] firstNames = new string[] { "Rob", "Jim", "Joe",
                                         "Nigel", "Sally", "Tim" };
    string[] lastsNames = new string[] { "Smith", "Jones",
                          "Bloggs", "Miles", "Wilkinson", "Brown" };

    SalesDB newDB = new SalesDB(connection);

    if (newDB.DatabaseExists())
    {
        newDB.DeleteDatabase();
    }

    newDB.CreateDatabase();

    foreach (string lastName in lastsNames)
    {
        foreach (string firstname in firstNames)
        {
            //Construct some customer details
            string name = firstname + " " + lastName;
            string address = name + "'s address";
            string bank = name + "'s bank";
            Customer newCustomer = new Customer();
            newCustomer.Name = name;
            newCustomer.Address = address;
            newCustomer.BankDetails = bank;
            newDB.CustomerTable.InsertOnSubmit(newCustomer);
        }
    }

    newDB.SubmitChanges();
}
```

The first thing the method does is make a connection to the database:

```
Customers newDB = new Customers(connection);
```

The variable `newDB` now represents a connection to the database with the given connection string. The method can use this connection to issue commands to manage the database.

The next thing that the method does is check to see if the database already exists. If it does, the database is deleted.

```
if (newDB.DatabaseExists())
{
    newDB.DeleteDatabase();
}
```

An important principle of testing is that a test should always start from exactly the same place each time. If there is already a database in a file in isolated storage it will need to be cleared before the test data is added.

```
newDB.CreateDatabase();
```

The next thing the method does is create a new database which contains a set of empty tables.

Once we have our new database we can create a set of test customer values and add each one to the customers table.

```
foreach (string lastName in lastsNames)
{
    foreach (string firstname in firstNames)
    {
        //Construct some customer details
        string name = firstname + " " + lastName;
        string address = name + "'s address";
        string bank = name + "'s bank";
        Customer newCustomer = new Customer();
        newCustomer.Name = name;
        newCustomer.Address = address;
        newCustomer.BankDetails = bank;
        newDB.CustomerTable.InsertOnSubmit(newCustomer);
    }
}
```

These two nested loops work through the same set of first and last names we used last time. The major difference from the previous test data generator is that this time a completed customer is added to the database:

```
newDB.CustomerTable.InsertOnSubmit(newCustomer);
```

The `InsertOnSubmit` method is how we add new entries into a table. We are using the `CustomerTable` property of the new database. If the database contained other tables we could add items to them in the same way. Note that this is typesafe; in other words if we tried to add a different type to the `CustomerTable` the program would not compile.

The final, and perhaps the most important, part of the method is the call that actually commits the changes to the database:

```
newDB.SubmitChanges();
```

This is directly analogous to the `Close` method when using file streams. This is the point at which the changes that we have asked for will be committed to the database. Up until the `SubmitChanges` method is called the system may keep some of the changes in memory to speed things up. This makes good sense if the same data item is being updated repeatedly. Rather than changing the database file each time, the system will use copy in memory and work with that. Only when this method is called will the memory copies be written back to the database file.

If you don't submit your changes there is a good chance that this will leave the database in a mess. Which would be bad.

We now have a database which we can use in our application. We can also create a test database with values we can look at. The database context that we are going to use will live in the `App.xaml.cs` program file along with a reference to the currently active customer.

```
public SalesDB ActiveDB;
public Customer ActiveCustomer;
```

When the program starts running the variable `ActiveDB` is set to the database we are using.

```
ActiveDB = new SalesDB("Data Source=isostore:/Sample.sdf");
```

Note that the file path has a particular format and identifies a file in the isolated storage. We can make use of multiple databases in our program if we wish, each will be held in a different file. This database file format is that of a standard SQL

database. A PC based database program could read the tables out of this file and it could be managed by an SQL database editor. The reverse is also true, a Windows Phone application could make use of a database that was prepared on a different computer and loaded into the application.

Programmer's Point: Remember the database is actually just a file

We can use the fact that the database is a file to quite good effect if we want to pre-load tables of data into an application. By using the file copy program in chapter 5 you could drop an already populated database from the application into isolated storage, where it can be picked up and used by the application.

Binding a ListBox to the result of a LINQ Query

Next we want to see how we can use this database in our application. In the simple Customer Manager program we used a `List` to hold all the customers. We found that the `ListBox` display element can take a list and display it. Now we want to take data out of the database and display this.

We get information out of the database by issuing queries. Previously we saw how the SELECT command could be used to fetch information. What we are now going to do is build a LINQ query to get all the customers out of the database.

```
var customers = from Customer customer
                in thisApp.ActiveDB.CustomerTable
                select customer;
```

This line of C# creates a variable called `customers`. The `customer` variable is of type `var`. If you've not seen this type before it can look a bit confusing. What `var` means in this context is "Get the type of this variable from the expression that is being put into it". This doesn't mean that the C# is giving up on strong typing, if we ever try to use the variable `customers` in an invalid way our program will still fail to compile.

The rest of the statement tells LINQ to get all the customer items from the `CustomerTable` member of the `ActiveDB` database context. This is returned as a list of customers that can behave as an `ObserableCollection`. Which is just what we need to give the `ListBox` to display the customers:

```
customerList.ItemsSource = customers;
```

At this point we have a fully working customer database. The only thing that we have to add to the program is code to submit the changes to the database when the user leaves the program. The best way to arrange this is to put the call into the `OnNavigatedFrom` method in `MainPage.xaml.cs`:

```
protected override void OnNavigatedFrom(
              System.Windows.Navigation.NavigationEventArgs e)
{
    App thisApp = Application.Current as App;

    thisApp.ActiveDB.SubmitChanges();
}
```

When the user moves away from this page the method will find the currently active database and submit any outstanding changes to it.

The solution in *Demo 01 SalesManagement* contains a Windows Phone application that implements a fully working customer manager. It does persist the data into isolated storage, but because the data storage is created fresh each time you will not see this. Once you have run the program once to create you can comment out the statement in App.xaml.cs that creates a new database and see this working for yourself.

Database and Data Binding Magic

I think this is actually quite magical. We have only added a small amount of code to our application and how we have the data elements stored and retrieved for us automatically. Changes that we make in the user interface are automatically being persisted in the database without any effort from us. It is worth just going through exactly what is happening here so that we can completely understand how it works. Consider the following sequence:

1. Users changes **"`Rob Miles`"** to "`Rob Miles the Wonderful`" in the `TextBox` holding the name field on the `CustomerDetailsPage`.
2. Because this `TextBox` is bound to the `Name` property in the `CustomerView` bound to this page it now updates that property in the `CustomerView` to the new name text.

3. User presses the `Save` button on the `CustomerView` page.
4. The `saveButton_Click` method in the `CustomerDetailsPage` runs and calls the `Save` method in the `CustomerView` instance behind this page.
5. The `Save` method copies the page properties from the view class into the active customer record.
6. Changes to the `Name` property in the active customer fires the property changed events in LINQ informing the database that the values are being changed.
7. LINQ flags the customer information as having changed and also raises a change event in the `ObservableCollection` which holds our customer list and is connected to the `ListBox` on MainPage.
8. The program returns to the `ListBox` view on `MainPage` and the display of the name "`Rob Miles`" is updated to "`Rob Miles the Wonderful`".
9. When the user exits the application the `OnNavigatedFrom` event fires and calls `SubmitChanges` on the database, storing the changes back into isolated storage.

If you don't believe this works, then try it and find out. You can use the above pattern anywhere you want to automatically save and load data with very little effort.

Adding Filters

By slightly changing the format of the LINQ query to fetch the data from the database we can select records that match a particular criteria:

```
var customers = from Customer customer
                in thisApp.ActiveDB.CustomerTable
                where customer.Name.StartsWith("S")
                select customer;
```

This query selects only the customers whose name starts with the letter S.

Figure 6-1 Filtered customer list

Figure 6-1 shows the result of such a query. This makes it very easy to extract particular elements from the database. We could easily add a search box to our application if we wanted to search for customers with a particular name.

6.2 Creating Data Relationships with LINQ

We are now able to store a large number of customer items and change and update them by using data binding.

However, to make our sales management system work we will need to create other tables and also link them together. To refresh our memories this is the design for the customer table:

Customer ID	Name	Address	Bank Details
123456	Rob	18 Pussycat Mews	Nut East Bank
654322	Jim	10 Motor Drive	Big Fall Bank
111111	Ethel	4 Funny Address	River bank

We have created a class which contains the required rows and can use LINQ to persist this data in a database file in isolated storage. The **Products** table is very similar to the Customers table, in that it just holds a list of items with certain properties.

Product ID	Product Name	Supplier	Price
1001	Windows Phone 7	Microsoft	200
1002	Cheese grater	Cheese Industries	2
1003	Boat hook	John's Dockyard	20

The Products class is actually very similar to the Customer one; it just contains a set of data items. The **Order** table is the more complex one. It actually contains references to rows in the other two classes.

Order ID	Customer ID	Product ID	Quantity	Order Date	Status
1	123456	1001	1	21/10/2010	Shipped
2	111111	1002	5	10/10/2010	Shipped
3	654322	1003	4	1/09/2010	On order

Each row of the Order table describes a particular order that has been placed. It gives the Product ID of the product that was bought, and the Customer ID of the customer that bought it. In C# this would be easy to create:

```
public class Order
{
    public DateTime OrderDate;
    public int Quantity;
    public Customer OrderCustomer;
    public Product OrderProduct;
}
```

The `Customer` and `Order` references would connect an order to the customer that made the order, and the product that was ordered.

However, when we are using a database we don't have any references as such. Rather than having a reference to a customer and product, instead the Order table holds the ID of the items that are related to each order. We then use the value in the Customer ID column to find the customer who has placed the order. For example, we know that order 1 was placed by Rob Miles because it contains a Customer ID of 123456, and so on.

In the good old days before computers this was how things worked. A company would have a filing cabinet full of customer data, another full of product data and a third full of orders. To find out the details of a particular order a clerk would have to look up the customer and product information from information written on the order details. The relational database provided a way for computers to manage information that contains relationships like these, what we want to do now is take the database relationship information and use it to create object references that make it easy to manipulate in an object oriented program.

LINQ Associations

In LINQ a relationship between one table and another is called an Association. This is implemented in the database classes by an `EntityRef` value.

Linking an Order to the Customer that placed it

From our system requirements the `Order` class needs to hold a reference to the `Customer` who placed the order. This is implemented as a relationship between two tables. The `EntityRef` is a special LINQ class we can use to connect two tables together, using a database to provide the underlying storage.

```
[Table]
public class Order : INotifyPropertyChanged,
                     INotifyPropertyChanging
{

    ...

    private EntityRef<Customer> orderCustomer;

    [Association(IsForeignKey = true, Storage = "orderCustomer")]
    public Customer OrderCustomer
    {
        get
        {
            return orderCustomer.Entity;
        }
        set
        {
            NotifyPropertyChanging("OrderCustomer");
            orderCustomer.Entity = value;
            NotifyPropertyChanged("OrderCustomer");
        }
    }
}
```

The `EntityRef` acts as a kind of glue between a simple reference (which we would like to use) and a lookup in a table (which is what LINQ will have to do when we use this reference). The good news is that as programmers we can write:

```
Customer newCustomer = new Customer();
Order newOrder = new Order();
newOrder.OrderCustomer = newCustomer;
```

This simple assignment will have the effect of connecting the order to the customer, so that the database record for that order holds the id of the customer the order is for. We can use this technique every time we want to implement a relationship between a row in one table and a row in another.

When we define the association property we also tell LINQ two other things. We tell it that the association is a "foreign key" and we also identify the property of the class that is going to store the value. If you have used databases before you will be familiar with the idea of a foreign key. This is a primary key from another database. In this case it is the primary key from the Customer database that can be used to identify the particular customer who placed the order.

The storage item in the association tells LINQ which private property of our database will actually hold the property information.

Databases and Navigability

Navigability in database systems is important. It allows a system to find one piece of information given another. At the moment we can start from an order and directly find the customer for that order. This is because a row in the order table contains the `CustomerID` of the customer for that order. However, we can't find our way back. Starting with an order we have no way of finding out the customer who placed it.

To allow proper navigability we have to put something in the `Customer` class that will allow us to find the orders that the customer has placed. However, things now get even more complicated because a customer could create lots of orders. This means that the nature of the relationships changes depending on which direction you are travelling:

- A particular order will only ever be associated with a single customer, the customer who created the order. This means that the relationship travelling from order to customer is one to one (one order to one customer).
- A particular customer will be associated with many orders. This means that the relationship travelling from customer to order is one to many (one customer to many orders)

When we design our databases we need to consider the navigability, in particular we need to think about how we are going to navigate using our relationships, and whether they are one to one or one to many.

Linking a Customer to all their orders

A customer may place many orders and so the relationship that we need to implement is a "one to many" relationship. If we were implementing this using C# classes we would use a collection of some kind, perhaps an array or a List. In LINQ such a relationship is provided by the `EntitySet` class. This is a bit like the `EntityRef`, except that it can manage a set of items, rather than just one. We can link a customer to their orders by adding an `EntitySet` to the `Customer` class.

```
[Table]
public class Customer : INotifyPropertyChanged,
                        INotifyPropertyChanging
{

    [Column(IsPrimaryKey = true, IsDbGenerated = true,
                                 AutoSync = AutoSync.OnInsert)]
    public int CustomerID { get; set; }


    private EntitySet<Order> orders = new EntitySet<Order>();

    [Association(Storage="orders", ThisKey="CustomerID",
                                   OtherKey="OrderCustomerID")]
    public EntitySet<Order> Orders
    {
        get
        {
            return orders;
        }
        set
        {
            orders = value;
        }
    }
}
```

This looks very like the `EntityRef` that we created earlier, except that it has extra information in the Association to describe the relationship.

The item `ThisKey` gives the name of the property that will be used by the `Order` to find the customer it is associated with. We can use `CustomerID`, the primary key for the `Customer` to do this.

The item `OtherKey` gives the name of the element in the `Order` class that will implement the association in the other direction, i.e. allows an order to find the customer it is related to.

You may be wondering why we have not fired any `NotifyPropertyChanged` events when `Orders` is changed. This is because changing the value of the `EntitySet` is something that we would hardly ever do. Remember that this is the container for the orders, not the orders themselves. We will add orders to the customer by putting things into the `EntitySet`, not by replacing the `EntitySet` itself.

To make the association work properly we have to make some changes to the reference in the `Order` class. These will bind both ends of the association together.

```
[Table]
public class Order : INotifyPropertyChanged,INotifyPropertyChanging
{

    ...

    [Column]
    public int OrderCustomerID;

    private EntityRef<Customer> orderCustomer =
                                        new EntityRef<Customer>();

    [Association(IsForeignKey = true, Storage = "orderCustomer",
                                         ThisKey = "OrderCustomerID")]
    public Customer OrderCustomer
    {
        get
        {
            return orderCustomer.Entity;
        }
        set
        {
            NotifyPropertyChanging("OrderCustomer");
            orderCustomer.Entity = value;
            NotifyPropertyChanged("OrderCustomer");
            if (value != null)
                OrderCustomerID = value.CustomerID;
        }
    }
}
```

The Association is defined as holding a foreign key (the primary key of the customer database) and using the value of `OrderCustomerID` to hold this value.

This version of the reference makes a copy of the `CustomerID` value into the `OrderCustomerID` column in the table when a customer is assigned to the order. In other words, if I write something like this:

```
Customer c = new Customer();
Order o = new Order();
o.OrderCustomer = c;
```

When the `OrderCustomer` property is assigned the set method above will take the `CustomerID` value and copy it into the `OrderCustomerID` value to record the customer for this order.

If all this is hurting your head a bit, and I must admit it hurt mine first time round, remember the problem that we are trying to solve. The `ThisKey` and the `OtherKey` descriptions in the association tell each side of the relationship how to find each other.

The good news is that once we have implemented this pattern we can easily add new orders to a customer:

```
Customer c = new Customer();
Order o = new Order();
o.OrderCustomer = c;
c.Orders.Add(o);
```

The `EntitySet` class provides an `Add` method that lets us add orders to the database. To work through the contents of the entity set we can use any of the constructions we would normally use to process collections:

```
foreach (Order o in c.Orders)
{
    // do something with each order
}
```

This doesn't look that special, but the special thing is that we are writing C# to work with the database.

Orders, Products and Database Design

We now have a link between customers and orders which works property. A customer can have a very large number of orders, and an order is always connected to the customer who placed it. The next thing we need to do is fill in the order with the products that the customer might have bought. This is the point at which we hit a snag with our original design of our `Order` table:

Order ID	Customer ID	Product ID	Quantity	Order Date	Status
1	123456	1001	1	21/10/2010	Shipped
2	111111	1002	5	10/10/2010	Shipped
3	654322	1003	4	1/09/2010	On order

If you look carefully at the design it turns out that there is no way we can have an order that holds more than one product. A particular order can be for multiple items, but they must all be of the same item.

It might be that this is perfectly OK. The person who wants to use your system might be selling items that are very expensive and are only ever ordered one at a time. Alternatively this might be disastrous; the system may be used by a supermarket that wishes to assemble orders which contain a huge number of different products.

To solve the problem we need to add another table, which we will hold all the items in each order. Each `OrderItem` will be linked to a particular order. Each `Order` will then hold a collection of these items.

OrderItem ID	Order ID	Product ID	Quantity
1	56	1001	1
2	56	1002	5
3	12343	1003	4

The table above shows that the order 65 was made up of two products, a Windows Phone and 5 Cheese Graters. (If you are wondering where the products come from, take a look at the Products table way back at the start of this chapter).

Database designers like to use diagrams to show how the various elements fit together. The diagram for the database that we have built looks like this:

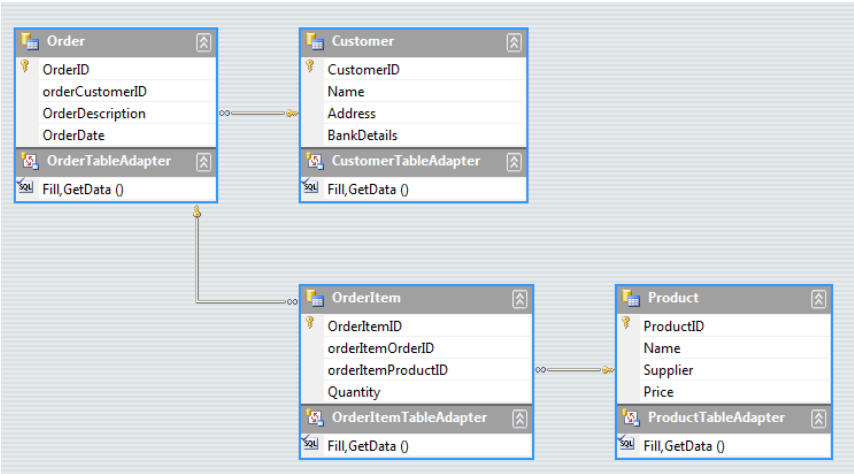


Figure 6-2 Database tables and their associations

The little keys show where a foreign key is being used at one end of the association. The little infinities (∞) show that an association can connect to any number of that item.

If you do a lot of database design you will be used to drawing diagrams like the one above. It was actually created from the database that was formed by the classes that we created.

Programmer's Point: Everything should be stored precisely once

When you design a database you should make sure that data is only stored in one way. In other words, we should not have a property "CustomerName" in the Order in Figure 6-2. People might suggest these kinds of things "to improve performance" but they are actually very dangerous. As soon as you store things in more than one place you have the problem of being **absolutely** sure that they are always synchronised. As another example, you should store the age of a customer or their date of birth, but not both. Otherwise you run the risk of users complaining that their age is sometimes wrong on the system.

Figure 6-3 Showing a list of orders

The solution in *Demo 02 SalesManagement* contains a Windows Phone application that adds Order, OrderItem and Product tables to the customer database. You can navigate the lists of customers and products and also find the orders that a customer has created. As an exercise you might like to add a list of the contents of each order.

LINQ Queries and Joining

Now that we have our data in a database we can use the very powerful query mechanisms to get information from it. For example, we might want to find all the orders that were placed on a particular date.

```
DateTime searchDate = new DateTime(2013, 8, 19);

var orders = from Order
                in activeDB.OrderTable
                where order.OrderDate == searchDate
                select order;
```

This query searches the order table of the currently active DB and finds all the orders created on the 19th of August 2013. We don't have to write any of the searching; the result is just returned in the variable called `orders`. You might be wondering what the actual type of `orders` is. Actually it is of type "LINQ query". The statement above doesn't do anything until you try to consume the data that it holds. For example, if we tried to work out the total value of the sales for that day we could write something like this:

```
int totalSales = 0;

foreach (Order order in orders)
{
    foreach (OrderItem item in order.OrderItems)
    {
        totalSales += item.OrderItemProduct.Price;
    }
}
```

The first `foreach` works through `orders` getting each order in turn. It is only when the program starts to actually consume the data that LINQ will leap into action and start producing results.

Note that this has implications if you want to use the result of a query more than once. If you use the same query twice you will find that your program will run a bit more slowly, as LINQ will be fetching things from the database twice. If you

want to use a result more than once it is best to use the query to product a list of items that you can then work through as many times as you like:

```
List<Order> DateOrders = orders.ToList<Order>();
```

This creates a list called `DateOrders` which contains the result of the query. We can use the list in our program without generating any further transactions on the database. The `ToList` method applies the query and generates a list of results.

Joining two tables with a query

LINQ has another trick up its sleeve. It can create queries that go across more than one table, a process called "joining". Perhaps we might want to create a complete list of all the orders made by all customers, listed in customer order. We want to find `Order` and `Customer` pairs that match up and list them. Using the `join` keyword LINQ can do this in one action:

```
var allOrders = from Customer c in newDB.CustomerTable
        join Order o in newDB.OrderTable on
                        c.CustomerID equals o.OrderCustomerID
        select new { c.Name, o.OrderDescription };
```

If you haven't seen one of these you would be forgiven wondering how on earth it compiles. The key is in the name LINQ, in that the `from`, `join` and `select` keywords are all integrated into the C# language to make these kinds of queries work. The first part of query finds all the customers in the customer table. This query is then joined onto a second, which works through all the orders doing a join on the two ID values stored in customer and order.

Essentially what it is saying is "Find me pairs of orders and customers where the `orderID` in the customer matches the customer id in the order".

Once it finds a match it then creates a new type that just contains the results that we want, in this case the `Name` from the customer and the `OrderDescription` from the order. Note the magic `var` is here again. In this case we get an SQL query that generates a collection of type containing two strings of the appropriate names.

We could use this query to produce a list of order descriptions by working through it

```
List<String> OrderDescriptions = new List<String>();

foreach (var orderDesc in allOrders)
{
    OrderDescriptions.Add(orderDesc.Name + " order: " +
                          orderDesc.OrderDescription);
}
```

As with the previous query, LINQ doesn't actually do anything until the program starts to work through the results. Note that we need to use a `var` type for the `orderDesc` value that we are fetching from the query. This contains the two items that we added when we created the type in the query.

The result of his query could be bound to a display element to show the list:

```
Rob Miles order: Camera Order
Rob Miles order: Cheese Order
Rob Miles order: Barge Pole Order
```

The var keyword

The `var` keyword can seem a little scary if, like me, you are used to declaring variables of a particular type and then just using them. I like the way that the C# compiler fusses about my programs and stops me from doing stupid things like:

```
int i = "hello";
```

An attempt to put a string into an integer is a bad thing to do, and will not compile. However, at first glance the `var` keyword looks like a way to break this lovely safety net. We can create things that don't seem to have a type associated with them, which must be bad. However, you can rest assured that although we don't give a name for a type like this, the C# compiler knows exactly what a `var` type is made from, and will refuse to compile a program that tries to use such a type in an incorrect way.

Deleting Items from a Database

We have seen how to add items to a database. It is also possible to delete items as well. For example, if the customer decided that they didn't want a particular order item we can delete it from the database:

```
ActiveDB.OrderItemTable.DeleteOnSubmit(item);
```

We can also delete collections of items using `DeleteAllOnSubmit`, including a collection identified as a result of a query.

Foreign Key Constraints and Delete

Some of our data items have relationships with others, for example a Customer will contain a number of Orders, and an Order will contain a number of OrderItem values. If the parent object is deleted before the children (i.e. if we try to delete a customer which contains some orders) the database will refuse to perform this, and will throw an exception as a result. If you want to delete a customer you will first have to delete all the order items in each order, and then delete the orders, and then delete the customer.

What We Have Learned

1. Windows Phone applications can use Language Integrated Query (LINQ) to interact with databases can be stored in files in isolated storage in the device. The databases themselves are managed by an SQL database server, but it is not possible to use SQL commands to interact with a database, instead queries are expressed using LINQ.
2. A database is made up of a number of tables. A table is made up of columns (different data items such as name, address, and bank details) and rows (all the information about a particular item – the name, address and bank details of an individual customer).
3. One column in a table acts as the "primary key", holding a value which can uniquely identify that row (the unique ID of that particular customer). The database can be asked to automatically create primary keys that are unique for each row in a table.
4. Windows Phone applications can create classes which are stored by LINQ as database tables. The attributes [Table] and [Column] are used within the class definition to specify the role of the components. Additional information can be added to the attributes to identify primary keys.
5. Database classes managed by LINQ can add notification behaviours to the data properties so that LINQ will automatically update the database entries when the properties are changed. If these are used with changed events in XAML display elements this can allow automatic database update when items are changed by the user.
6. A database can contain relationships, where a column in one table contains primary key values which identify rows in another. These are called "foreign keys" as they are primary keys, but not for the table which contains them.
7. LINQ implements relationships by using the EntityRef class, which contains a reference to a foreign key in another database. This is used for "one to one" relationships. The EntitySet class is used to allow a member of a table to refer to a large number of rows in another table. This is used for "one to many" relationships. These data elements are modified with the Association attribute that describes how the table relationship is implemented.
8. A database will frequently have a one to one going in one direction (an order is attached to one customer) and a one to many going in the other (a customer can place many orders). For such associations to work correctly is important that the associations are correctly configured and that foreign keys are stored correctly.
9. A LINQ query can be used to fetch structured data from the database. The query is only actually executed when the results of the query are being consumed by the program. LINQ queries can use the join keyword to combine the results of multiple queries.
10. When deleting items from a database all child items must be deleted before the parent object is removed, otherwise there would be elements in the database containing primary key values for items that don't exist.

7 Networking with Windows Phone

A networked device is much more useful than one that is just free standing. Once a system is connected to the internet all kinds of things become possible. Mobile phones are around the most connected devices there are, in that they have a whole variety of connection possibilities. However, they also bring challenges to the application developer in that any, or perhaps all, of these network connections could be removed at any instant, and the user of our application will expect it to handle this automatically.

In this section we will look at the network connection options available to programs that we write. We will also give some consideration to how we can make sure that our applications can respond gracefully to network problems.

7.1 Windows Phone Network Support

All Windows Phones support network connectivity via the mobile telephone network and also by means of their built in WIFI. As far as our programs are concerned the two network technologies are interchangeable. The programs we are going to write will initiate connections to a data service and the underlying phone systems will make the connection the best way that they can.

Of course, if there is no mobile signal or WIFI available these connections will fail. Applications that we write must be able to deal with this situation gracefully. At the very least this means displaying a "Sorry we can't connect you just now, please try later" message. A more advanced program may store data locally and then send it later. It is up to you how your program will work in this respect.

Another issue concerns the actual network connection that is used. On a mobile device it is much more expensive to use the cellular telephone network than it is to transfer data using Wi-Fi. Many phone users are on contracts which "cap" the amount of data that can be transferred over the telephone network in any given period. If they use more than their allocated amount this can be hugely expensive. If we create an application which begins to download or stream large amounts of data without consideration of the cost of the currently available network connection this will not go down well with users.

The Windows Phone Emulator

The Windows phone emulator has the same networking abilities as the real device. It operates as a completely separate device on the network, totally separate from the host PC it is running on. The emulator has its own *media access control* (mac) and *internet protocol* (it) address (we'll talk about these later). Note that this is a change from earlier versions of the Windows Phone emulator, which were implemented as programs that ran as a program and used the same network connection as the host PC. The good news is that this change makes the network environment of the emulator a lot more realistic, the bad news is that it can be slightly more difficult to configure, as we shall see later.

However it is also important to test programs on a real device. On a real Windows Phone we can disable network services and test our applications to ensure that they perform properly over lower speed connections and when the network connection is suddenly removed.

7.2 Networking Overview

Before we look at how Windows Phone uses network connections, we need to learn a little bit about networks. This is not a detailed examination of the field, but it should give you enough background to understand how our programs are going to work.

Managing the movement of data

The first computer networks used wires to send their data signals, although more modern networks can use radio or fibre optic cables. Whatever the medium is, the fundamental principle is that you have some hardware that can put data onto the medium in the form of bits and get it off again. A bit is either 0 or 1 (or you can think of a bit as either true or false) and can be signalled by the presence or absence of a voltage, a light from a light-emitting diode (LED), or a radio signal.

If you imagine signalling your friend in the house across the road by flashing your bedroom light on and off, you have an idea of the starting point of network communications.

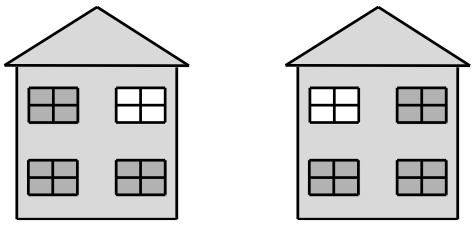


Figure 7-1 House to house networking

Once we have this raw ability to send a signal from one place to another, we can start to transfer useful data. We will invent a protocol (an arrangement of messages and responses) and then use this to pass messages.

To communicate useful signals you have to agree on a message format. You could say, "If my light is off and I flash it twice, it means it is safe to come round because my sister is out. If I flash it once, it means don't come, and if I flash it three times, it means come and bring pizza with you." This is the basis of a thing called a protocol, which is an arrangement by parties on the construction and meaning of messages.

If you think about it, the messages and the protocol are actually independent of each other. We could replace "flash the light" with "tap on the water pipes" and the protocol could be the same. Three flashes or three taps could both mean "bring pizza".

When we are designing networks we can express this by using layers:

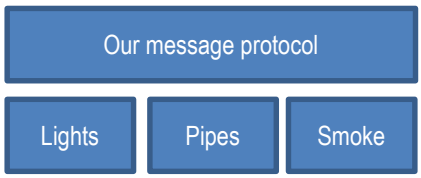


Figure 7-2 House networking protocol layers

The protocol, for example "three for pizza" sits on top of a "hardware" layer that can deliver the network events. In Figure 7-2 we can use light flashes, bangs on the pipe or even puffs of smoke to deliver network messages.

Each low level layer will contain standards like "a flash must be no longer than half a second" and all the flashes must occur within a two second period". The message protocol on top will be designed with no consideration of how the messages are sent, it will just be told that three events have been received.

This is really flexible. It means that we can add new kinds of message delivery, for example flag waving, without having to change the entire network. The network protocols used in by the Windows Phone are based on this layered approach.

Network layers in Windows Phone

In the case of Windows Phone there are a whole range of signal types that it can use to connect with a network. There is the WIFI connection, the mobile phone cellular connection and even the Bluetooth connection to the Windows Phone software on the PC. All of these technologies can be used to pass data between programs on the phone and the Internet.

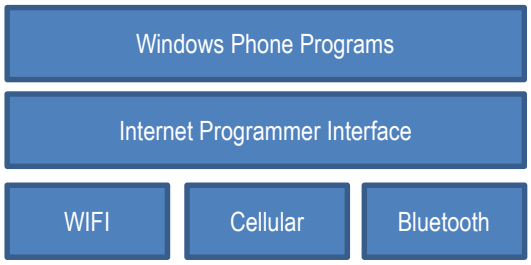


Figure 7-3 Windows Phone networking layers

If you look at Figure 7-3 you will notice that I've added another layer on top. This is the actual programs that we will write. The Windows Phone programs talk to the Internet Programmer Interface (the thing we are going to learn how to use) and the Internet Programmer Interface talks to the three connection technologies.

The good news is that the programs we write don't need to be changed to work over the different types of connection. However, there are things that we need to remember, in that if our phone user goes abroad they may not allow data roaming, so the cellular data connection will be disabled. Even worse for our programs, a user may put their phone into "aircraft" mode and disable all network connections for a while.

Finding available network connections

A Windows Phone program can determine which connections are available by using methods in the `DeviceNetworkInformation` class.

```
using Microsoft.Phone.Net.NetworkInformation;

...

System.Text.StringBuilder sb = new System.Text.StringBuilder();

sb.Append("Operator: " );
sb.AppendLine(DeviceNetworkInformation.CellularMobileOperator);

sb.Append("Network available:  ");
sb.AppendLine(DeviceNetworkInformation.IsNetworkAvailable.ToString());

sb.Append("Cellular enabled:  ");
sb.AppendLine(DeviceNetworkInformation.IsCellularDataEnabled.ToString());

sb.Append("Roaming enabled:  ");
sb.AppendLine(DeviceNetworkInformation.IsCellularDataRoamingEnabled.ToString());

sb.Append("Wi-Fi enabled:  ");
sb.AppendLine(DeviceNetworkInformation.IsWiFiEnabled.ToString());
```

The above code assembles a message that describes the status of the phone which can then be displayed. We can also use these values to control the way that our application behaves.

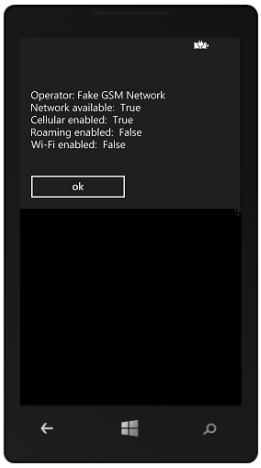


Figure 7-4 Available networks

The solution in *Demo 01 NetworkDiagnostics* contains a program that displays the network status for a phone or emulator.

Programmer's Point: Everything should be stored precisely once

When you design a database you should make sure that data is only stored in one way. In other words, we should not have a property "CustomerName" in the Order in Figure 6-2. People might suggest these kinds of things "to improve performance" but they are actually very dangerous. As soon as you store things in more than one place you have the problem of being **absolutely** sure that they are always synchronised. As another example, you should store the age of a customer or their date of birth, but not both. Otherwise you run the risk of users complaining that their age is sometimes wrong on the system.

Building Up to Packets

Just flashing your light to your friend willy-nilly does not allow you to send much information. To communicate useful signals you have to agree on a message format. You could say, "If my light is off and I flash it twice, it means it is safe to

come round because my sister is out. If I flash it once, it means don't come, and if I flash it three times, it means come and bring pizza with you." This is the basis of a thing called a protocol, which is an arrangement by parties on the construction and meaning of messages.

A modern network sends data in 8 bit chunks that are called *octets* by the people who design them. In C# we can hold the value of a single octet in a variable of type `byte`. If we have 8 data bits we can arrange them in 256 different patterns, from 00000000 (all the bits clear) to 11111111 (all the bits set). An octet is the smallest unit of data that is transferred by a network. A give octet might be a number (in the range 0 to 255) a single character (a letter digit or punctuation) or part of a larger item, for example a floating point number or a picture file. On the internet a data packet can be several thousand octets in size.

Addressing Packets

Your bedroom light communication system would be more complicated if you had two friends on your street who want to use your bedroom light network. You would have to agree with them that you would send two sets of flashes for each message. The first one would indicate who the message was for, and the second would be the message itself. Computer networks function in exactly the same way. Every station on a physical network must have a unique address. Packets sent to that address are picked up by the network hardware in that station.

Networks also have what is called a broadcast address. This allows a system to send a message which will be picked up by every system. This is the network equivalent of "Calling all cars . . ." In our communication network, this could be used to warn everyone that your sister has come home and brought her boyfriend, so your house is to be avoided at all costs. In computer networks a broadcast is how a new computer can find out the addresses of important systems on the network. It can send out a "Hi. I'm new around here!" message and get a response from a system that can then send it configuration information.

Everyone on a particular network can receive and act on a broadcast sent around it. In fact, if it wanted to, a station could listen to all the messages traveling down its part of the wire or Wi-Fi channel. This illustrates a problem with networks. Just as both of your friends can see all the messages from your bedroom light, including ones not meant for them, there is nothing to stop someone from eavesdropping on your network traffic. When you connect to a secure Web site, your computer is encoding all the messages that it sends out so that someone listening other than the intended recipient would not be able to learn anything.

Internet Addressing

We have already seen the networks we are using can send 8 bit sized items. This would mean that if we used a single octet to address stations on the network we could not connect more than 256 systems. To get around this the present version of the Internet (IP V4) uses four 8 bit values which are strung together to allow 32 bits to be used as an address. In theory this allows for over four thousand million different systems on the internet. However, this large number is hard to achieve in practice and so a new version of the Internet (IP V6) is being rolled out which has 128 bit addresses (16 octets in each address). This would allow for a huge increase in the number of possible stations, but since it is a fundamental change to way the network functions it is taking a while to put into practice. At the moment the networking software in the Windows Phone does not support IPV6 addressing.

Routing

Not everything on the internet is connected to the same physical network. Signals sent around the wires in my house to not make as far as the network operated by my next door neighbour. Instead we have to view the Internet as a large number of separate networks which are connected together. This means that to get messages from one physical network to another we have to introduce the idea of routing.

If you had a friend on the next block, she might not be able to see your bedroom light. But she might be able to see the light of your friend across the road. This means that you could ask your friend across the road to receive messages and then send them on for you. Your friend across the road would read the address of the message coming in, and if it was for your friend on the next block, she would transmit it again. Figure 16-2 shows how this works. Your friend uses the window on the left to talk to you and the window on the right to relay messages to your more distant friend.

Figure 7-5 Routing round the houses

You can think of your friend in the middle as performing a routing role. She has a connection to both "networks", the people you can see and also the people that your distant friend can see. She is therefore in a position to take messages from one network and resend them on the other one. Note that this is not like passing a letter from one person to another. Instead your friend is receiving your message and then re-transmitting it to your distant friend.

An address of a given system on the Internet is made up of two elements, the address of the local network containing the system and the address of that system on the network. The important thing to remember about a local network is that the machines in it are actually connected to the same physical media.

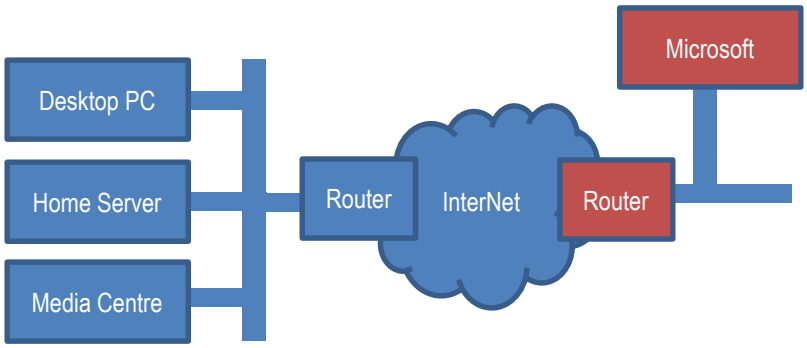


Figure 7-6

Figure 7-6 Routing and the internet

The diagram in Figure 7-6 shows how this works. The machines on your home network are physically connected and can talk directly to each other. The Desktop PC can fetch files directly from the Home Server. However, if the Desktop PC needs to contact Microsoft the messages must leave the local network and travel via the Internet. Just before the Desktop PC sends any message out it looks at the network address of the system it is talking to. If the remote system is on the same network it will connect directly the system. However if, like Microsoft, the system is on a different network the Desktop PC will send the message to the router which sends such messages into the internet, which delivers them to the router at the Microsoft.

Packets that you send from your home PC to distant machines are sent to the network at your Internet service provider (ISP) which then re-transmits them to the next system in the path to the destination. Packets might have to be sent over several machines to reach their destination. The Internet constantly changes the routes that packets take. This makes it very reliable and able to manage sudden surges in traffic and failures of parts of the network but it can lead to situations where a packet arrives before another which was sent first. Sometimes packets can get lost (although this is fairly rare), so you can't be sure that one has arrived until you receive an acknowledgement. One thing you should remember is that you do not really "connect" your system to the Internet. Whenever your system is connected, it actually becomes part of the Internet.

You can regard a local-level network as the same as the internal mail that is used in many organizations, including my university at Hull. If I want to send a message to our chemistry department, I just put the address "Chemistry Department" on the envelope and drop it in the internal mail. There is a local protocol (called the internal mail system) that makes sure that the message gets there. However, if I want to send a message to the chemistry department at York University, I must put a longer address on the envelope. When the letter gets to the mailroom the staff notice that the destination is not local, and they route it out to the postal system, which sends it to York. This is the "Internet Protocol" for letters.

The Internet is powered by a local protocol (Transport Control Protocol, or TCP) and an internetwork protocol (Internet Protocol, or IP). Put these together, and you have the familiar TCP/IP name that refers to the combination.

You can also use the TCP/IP protocol to connect machines without linking them to the Internet. If you plug your Windows Phone into your PC it actually creates a "tiny internet" between the two machines so that the Zune software can transfer media to and from your phone.

Anything that you send via the Internet will be transferred using one or more individual packets. Each packet contains the address of the destination and each packet is numbered, so that missing packets can be detected and packets can be put into the right order when they are received (if you want that). If you need to transfer a large file, this will be broken down into a number of packets.

Networks and Security

If we go back to the original "bedroom light" based network we can see that it is inherently insecure.

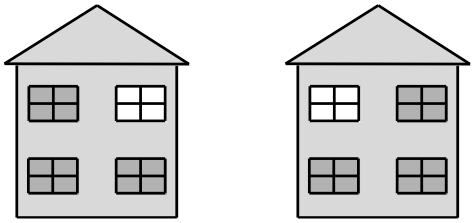


Figure 7-7 Eavesdropping

Anyone in the street outside the houses can see the lights flash and eavesdrop on our conversations. In the case of ordering pizza this is probably not a problem, but it is a serious problem on the internet because a lot of information that is transferred is confidential. The problem is solved by using *encryption*. The data is not sent in "raw" form, instead it is passed through an encryption stage before being sent. Then the receiver then runs received data through a decryption stage before acting on it.

Unless the eavesdropper knows what kind of code is being used and the keys that are being used to encrypt the data they will not be able to make sense of what is sent. Security of this kind can be added to data so that it can be carried securely. However, it is important to remember that if you send "raw" data it will be possible for anyone in the signal path to extract and read your information. Websites can use an encrypted version of the Hyper Text Transfer Protocol (HTTP) called HTTPS which allows them to exchange secure data. A Windows Phone can use this to secure web requests that it makes to clients.

Note that this security is not at the level of authenticating you as a user; it is to make sure that your computer is connected to the web site that it thinks it is, and then nobody can read the messages that are exchanged.

Networks and Media

Note that links between different parts of a network can use different media. A laptop could connect via WIFI to a home network which would be connected via the service provider to the Internet backbone. The internet backbone is connected to your mobile phone provider which then sets up a network connection from a cell phone tower to your Windows Phone. This is how you can use your Windows Phone to message your friends. Furthermore many devices have multiple network connections. Your phone can talk to the Internet via a cell connection to your mobile phone provider or it can use your home WIFI. The good news as far as our programs are concerned is that the way we make connections is independent of the precise connection.

Networks and Protocols

A protocol is a set of rules that tells you how to behave in a certain situation. There is a protocol that tells you which knife and fork to use in a posh banquet and another that tells you to kiss a maiden on her hand having just rescued her from a dragon. You already have one with your friend, where you have agreed on the meaning of the various messages that you send with your bedroom lights.

In networking terms, a protocol sets out the design of all the messages and how stations in a network should cooperate to move data around. There are essentially two levels at which this must take place. There must be a "local-level" protocol that lets local stations (ones on the same piece of physical media) exchange data, and there must be an "internetwork" protocol that allows messages to be sent from one local network to another. The protocol should also extend to cover how systems on the network are addressed and discovered by other systems.

The Internet is based on a large number of different protocols which all work together to describe how systems can work together to exchanged data.

Finding Addresses on the Internet using DNS

The Internet addresses everything by their 32 bit network/system values, which are usually expressed as four "octet" values. For example the address on the internet of the University of Hull web server is `150.237.176.24`. This 32 bit value is what Internet messages use to find their way to the web server system at our university. An address is split into `150.237` which is the address of the university network in the world and `176.24` which is the address on the Hull network of our web server. This form of a network address is called the Internet Protocol or *IP* address of a system.

However, nobody wants to have to remember an IP address like this just because they want to see our university web pages – beautiful though they are. People would much rather use a name like `www.hull.ac.uk` to find the site. To solve this problem a computer on the Internet will connect to a *name server* which will convert hostnames into IP addresses. The system behind this is called the *Domain Name System*. It is a collection of servers who pass naming requests around amongst themselves until they find a system with authority for a particular set of addresses who can match the name with the required address. We can think of a name server as a kind of "directory enquiries" for computers. In the old days if I wanted to know the phone number of the local movie house I would ring up directory enquiries who would tell me this. When a computer wants to know the IP address of a web site a user wants to visit it will ask its Domain Name service to provide the result.

Addresses and Subnets

You might think that if every device on the internet has an IP address it should be possible to uniquely address that specific device. In other words, if you get hold of the IP address of my phone you should be able to send network messages directly to it. After all, every phone in the world has a unique number, doesn't it?

This is not actually the case though. Because of the worldwide shortage of IP addresses many networks are arranged into subnets. In this arrangement systems on the subnet have IP addresses that work on that subnet but are not visible to machines outside the subnet. The subnet router, which has an IP address on the "proper" internet, sends messages on behalf of all the machines on the subnet. When messages come from the internet the router performs some address translation to ensure the packet is sent to the correct local system. This arrangement works well, and is used by companies and home networks to reduce the number of "worldwide" IP addresses that they need. It is also used by mobile phone carriers. This rather like a company having an internal telephone system with many staff telephones and only a few external phone numbers. Calls to the company phone number are switched to particular phones inside the company.

Unfortunately this internet addressing limitation can make it difficult for two mobile devices to directly exchange data, as their addresses are not meaningful to each other. The only way to solve this problem is to use a third party machine which is on the wired network and has a "proper" IP address. Both phones can originate a connection to this third machine and use it to pass data backwards and forwards.

Of course, if the phones are both connected to the same WIFI network when you are at home it should be possible for them to connect directly as they will both be on the same physical network.

IPV4 and IPV6 Addresses

The addresses that I'm using in this text are Internet Protocol Version 4 addresses. In this addressing scheme each address is made up of 4 data bytes, giving the potential for around four thousand million addresses. At the time this addressing scheme was invented this seemed plenty, because the smallest networkable computer you could was around the size of a desk. Today however networked computers can be tiny, and there are many more of them than anyone ever expected.

Internet Protocol Version 6 (IPV6) has been designed address this lack of, er, addresses along with lots of other networking issues that are raised by the potential of many billions of network users. IPV6 has 128 bits in each network address allowing for many billions of addresses. Windows 8 supports IPV6 addresses, but earlier versions of Windows Phone do not.

For the purpose of this text I'm going to use IPV4, old style, addresses. The fundamental principles of operation are the same for both addressing schemes.

Networks and Ports

We now know how the Internet sends messages from one computer to another. Each computer has an address that identifies the internet the network the computer is connected to and the address of that computer on the network. Now we have to add another layer to the protocol, and consider ports.

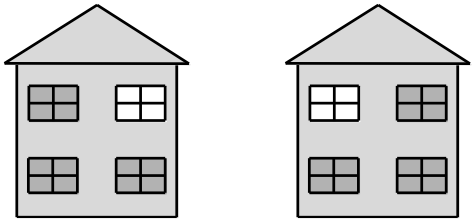


Figure 7-8 House to house messages

If we go back again to our original network, where we are communicating with our friend by flashing our bedroom light, we have a problem if we want to talk about different topics at the same time. If we wanted to send messages about what is coming up on TV, and also order food, we would have to improve our protocol so that some messages could be flagged as being about TV shows, and others identify the type of pizza we want.

In the case of a computer system we have exactly the same problem. A given computer server can provide a huge variety of different services to the clients that connect to it. The server might be sending web pages to one user, sharing video with another and hosting a multiplayer game all at the same time. The different clients need a way of locating the service that they want on from the server.

The Internet achieves this by using "ports". A port is a number which identifies a particular service provided by a computer. Some of the ports are "well known", for example port number 80 is traditionally the one used for web pages. In other words, when your browser connects to a web page it is using the internet address of the server to find the actual computer and then connecting to port 80 to get the web page from that server.

When a program starts a service it tells the Internet software which port that service is sitting behind. When messages arrive for that port the program is passed these messages. If you think about it, the Internet is really just a way that we can make one program talk to another on a distant computer. The program (perhaps a web server) you want to talk to sits behind a port on a computer connected to the Internet. You run another program (perhaps a web browser) that creates a connection to that port which lets you request web pages and display them on your computer.

Programmers can write programs that use any port number, but many networks use *firewalls* that only allow particular packets addressed to particular well know ports to be passed through. This reduces the chance of systems on the network coming under attack from malicious network programs.

Connections and Datagrams

The Internet provides two ways systems can exchange information: connections and datagrams. A datagram is a single message that is sent from one system to another, in the same way that we could just flash the lights in our inter-house network to deliver a message to someone who may or may not be watching. This is like flashing your bedroom light three times to ask for pizza and then just waiting for someone to turn up with your favourite pepperoni thin and crispy.

If you want to be sure that a message has got through you could agree with your friend that she would flash her light once to indicate that she has received it. Then perhaps you could send another message asking for drinks. When she was leaving to fetch the pizza, she could flash her light twice to indicate that she was "going off the air." This would be the basis of a *connection* between the two of you.

When two systems are in a connection, they have to perform extra work to manage the connection itself. When one system sends a message that is part of a connection the network either confirms the message was successfully transferred (once the network has received an acknowledgement) or gives an error saying that it could not be delivered.

Connections are used when it is important that the entire message gets through. When your browser is loading a Web page, your computer and the Web server share a connection across the network. This makes sure that all parts of the Web page are delivered and that any pieces that don't arrive first time are retransmitted. The effort of setting up and managing the connection and requesting retransmission when things fail to turn up means that data in connections will transferred more slowly. Managing a connection places heavier demands on the systems communicating this way.

The Internet has a protocol, called Transmission Control Protocol (or TCP), which describes how to set up and manage a connection between to network stations.

Datagrams are used for sending data in situations where you don't need an acknowledgement. If you were flashing your bedroom light every ten seconds to show the score in a football game it wouldn't matter if anyone receiving your messages missed a message as they would just have to wait for the next one to arrive. If a datagram is lost by the network, there is no point in asking for it again because by the time the replacement arrives, it will be out of date. Datagrams work well when

you want to send data as fast as you can and it doesn't matter if some gets lost on the way. They are used for streaming media, where moving video is being sent and the priority is to get the signal delivered as fast as possible.

The internet has a protocol called User Datagram Protocol (or UDP) that describes how systems can send datagrams between each other.

The Windows Phone networking system can send datagrams or create connections, depending on what you want to do. You need to decide which connection type you should use in a particular situation.

7.3 Addresses and Networks

We are going to start by creating a pair of programs that will use User Datagram Protocol (UDP) to transfer messages. One program will be the sender, and the other the listener. We can run both programs in different emulators on our PC and see them work, or we can use two phone devices and send messages from one to the other.

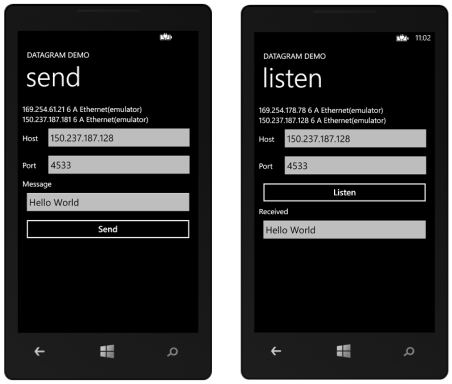


Figure 7-9 Datagram Demos

Figure 7-9 shows how this is going to work. The sender will type in the host and the port to use for the conversation. The listener will enter the address and port they are going to listen on and then display messages that arrive.

However, before we can start to send messages we first have to consider the issue of networks and addresses. This is of particular importance when we use the Windows Phone emulator.

A Windows Phone can have lots of network addresses. It can have at least one per network interface. Remember that one of the great things about the TCP/IP network protocol is that it will work on lots of different transports. Two programs can talk to each other using Wi-Fi or the cellular network and be unaware of the actual communications medium moving the data around on their behalf. Underneath this though, the device must have an address on each network.

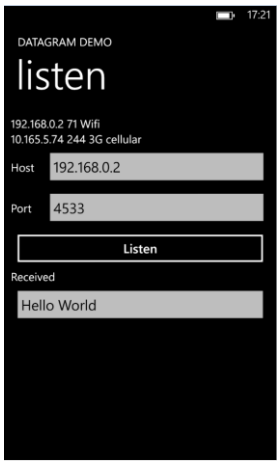


Figure 7-10 Windows Phone Device Network addresses

Figure 7-10 is a screenshot from a phone running the Network listen program that receives and displays datagrams. Note that at the top of the display you can see two network addresses. One is labelled Wi-Fi and is the IP address the phone has on the Wi-Fi network at my house. The other is labelled 3G cellular, and is the IP address that the phone has on the 3G network it is also connected to. These are two different *physical* networks, which use different hardware and different radio signals.

When we try to connect two programs it is very important that we make sure that they are using addresses that work on the correct physical network. If the sender is sending to the 3G network address and the listener is listening on their Wi-Fi address then they are not going to be able to contact each other. This would be like waiting for a phone call from your friend when in fact they have sent you a letter.

Private and internet IP addresses

It is very common for companies, mobile phone networks and even people at home to set up their own networks that use the TCP/IP protocol to connect. Each system that is connected to a network that uses the TCP/IP protocol must have a unique IP address on that network. However, this does not mean that having an IP address automatically makes a system accessible from the internet. Both of the addresses in the screenshot above are on "private" networks, not the internet. Devices connected my Wi-Fi network at home would be able to contact the phone using the Wi-Fi IP address.

If my phone needs to contact a machine on the global internet it could send the request to my home router, which does have an address on the Internet which it can use to transfer packets. The same thing happens with cellular network connection. This means that my phone never has an IP address on the internet itself.

However, if we want to make a game to play between two phones that are on the same Wi-Fi network they can connect to each other because they both have IP addresses that are on the same network. That is how our send and listen example programs are going to work.

Network addresses and the emulator

We know that the emulator runs within a Windows PC but is completely separated from the Windows PC itself. It uses a technique called "hardware virtualisation" which allows the processor in the PC to act as if it is more than one computer. The result is that the Windows PC is running the emulator does not really run a program that "pretends" it is a Windows Phone. It actually starts up a "computer within a computer" and makes that computer run the same program code that a real Windows Phone runs.

This means that you get a highly accurate simulation of the Windows Phone and it also means that programs in the emulator are completely unaffected by actions in Windows on the host PC, and vice versa.

It also means that the emulator will have an IP address which is completely separate from the underlying PC and that if you run two emulators they will each have their own IP address.

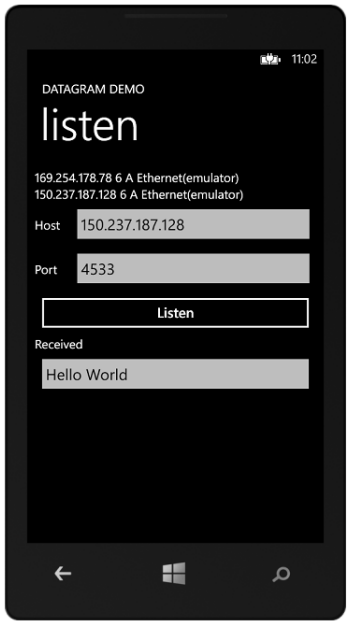


Figure 7-11 Windows Phone Emulator network addresses

Figure 7-11 shows a Windows Phone emulator running the same program as the Windows Phone device in Figure 7-10. Note that there are two networks, and both of them have network type 6, which means a wired network. This is kind of sensible, in that the emulator does not have a real cellular or Wi-Fi network, but the numbers look a bit confusing.

This is because when the emulator is installed it also installs a "virtual network" on your PC. This network has its own IP addressing scheme and contains a "virtual gateway" onto the outside world. The emulator will also attempt to claim IP addresses on the network the Windows PC is connected to, and use these so that the emulator can be addressed on the local network. In Figure 7-11 above you can see that one of the addresses of the emulator is `192.168.0.27`. This is actually on my network at home, and means that a Windows Phone on my home network can talk to this emulator by giving that address.

Unfortunately these addresses are set up when the emulator starts running and are not guaranteed to be the same each time, which is why my sample programs display them when the program runs, so that you can enter them on the device that wants to talk to them. You might actually see as many as four network addresses in the emulator, in this case you may need to try different ones until you find the one that works. This is issue is further complicated by role of *firewall* programs that might stop your PC from sending messages from emulators onto your local network.

Finding network addresses

The programs in Figure 7-10 and Figure 7-11 show a list of all the networks active on the phone and gives their "interface number". There is an internet standard which maps interface numbers to particular types of connect. For example, interface number 71 is Wi-Fi and interface number 244 is 3G Cellular.

```
Dictionary<uint, string> networkDecode = null;

string decodeNetworkNumber(uint number)
{
    if (networkDecode == null)
    {
        networkDecode = new Dictionary<uint, string>();
        networkDecode.Add(1, "Network");
        networkDecode.Add(6, "A Ethernet(emulator)");
        networkDecode.Add(9, "Token ring");
        networkDecode.Add(23, "PPP");
        networkDecode.Add(24, "Lopback");
        networkDecode.Add(37, "ATM");
        networkDecode.Add(71, "Wifi");
        networkDecode.Add(131, "Tunnel");
        networkDecode.Add(144, "Firewire");
        networkDecode.Add(244, "3G cellular");
    }
    if (networkDecode.ContainsKey(number))
    {
        return networkDecode[number];
    }
    else
    {
        return "unknown";
    }
}
```

The method `decodeNetworkNumber` will return the name of the network for a given network number. It uses a dictionary as a lookup table and fills in some of the more common numbers with matching descriptions. The dictionary takes in an integer and returns the string stored with that key.

```
ObservableCollection<string> IPAddressList =
                        new ObservableCollection<string>();

void showNetworks()
{
    IPAddressList.Clear();
    IPAddresesListbox.ItemsSource = IPAddressList;

    var ipAddress = NetworkInformation.GetHostNames();

    foreach (var name in ipAddress)
    {
        uint itype =
          name.IPInformation.NetworkAdapter.IanaInterfaceType;
        string text = name.RawName + " " + itype + " " +
            decodeNetworkNumber(itype);
        IPAddressList.Add(text);
    }
}
```

The code above works through each of the networks available on a given phone device and constructs a list of strings, with a string describing each network. It uses the `NetworkInformation.GetHostNames()` method which returns a collection of all the network hosts available. The `NetworkInformation` class is in the `Windows.Networking.Connectivity` namespace. If you try using the sample programs you will find it very useful to be able to get the various network hosts available for a device, as you will need to give these network address on the correct network.

The method above uses a `ListBox` called `IPAddresesListbox` which is connected to the `IPAddressList` collection and displays it. The `showNetworks` method is called when the page is navigated to.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    showNetworks();
}
```

This makes sure that the list of networks is up to date when the user arrives on the page.

Working with Network Addresses

The upshot of all this is that if you have two phones on the same network it is actually non-trivial to get them to talk to each other, as neither will know the address of the other, and you don't want to have your users typing in IP addresses to get things to work.

Fortunately there are ways in which two phones can connect and then exchange network addresses, later in the text we shall see how you could use a Bluetooth or Near Field Communication (NFC) connection to pass the IP addresses from one phone to another.

In the examples we are going to use you need to make sure that when you try to link to devices you use addresses which are on the same network for each device. In all the examples that we will be using the internet address of the phone is displayed, and you can enter the address of the destination device by hand.

7.4 Sending and Receiving User Datagram Protocol (UDP) messages

We are going to start by creating a pair of programs that will use User Datagram Protocol (UDP) to transfer messages. One program will be the sender, and the other the listener.

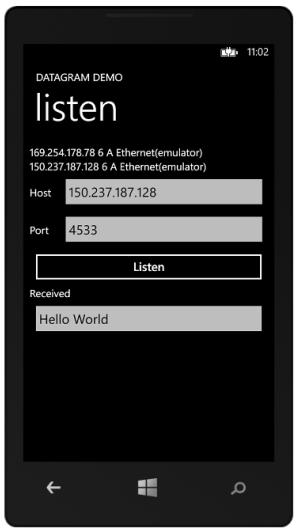


Figure 7-12 The Listen program running on the emulator

Figure 7-12 shows the listener program in use. The user will type in the address of the interface that the phone will listen on (note that this is the address of the interface on the listening phone). When we are using datagrams the listener will listen "promiscuously" on that port, in other words any system on the network can send data to it. The datagrams that are received do contain the address of the sender however, so a system can decide to ignore datagrams from certain senders if it wishes.

The Listener will listen on a particular port. By default the program above uses port number 4593. There is nothing particularly special about this number. Ports are numbered in the range 0 to 65,535 and the first 1,024 are "well known" numbers which are reserved for things like web browsers, email and the like. The port numbers above 1,024 can be used within a sub-net for whatever we like.

Programmer's Point: Only use special ports within a subnet

If you have two systems on the same physical network you can usually use whatever port numbers you like to exchange data between them. However, it is unlikely that the router that connects a physical network to the internet will pass datagrams addressed to strange ports as these are usually filtered out. The only port that is very likely to work is the port number 80, which is the port used for World Wide Web traffic.

When the user presses the Listen button the program creates a listener which is bound to the address and port that have been entered.

Creating a Listener

The object that is going to manage the incoming events is of type `DatagramSocket`:

```
DatagramSocket receiveSocket = new DatagramSocket();
```

This instance is created when the class is loaded. The program will call methods on it to set up and respond to the network messages.

```
async private void ListenButton_Click(object sender,
                                      RoutedEventArgs e)
{
    receiveSocket.MessageReceived +=
                          receiveSocket_MessageReceived;
    await receiveSocket.BindEndpointAsync(
                new HostName(HostTextBox.Text),
                PortTextBox.Text);
}
```

The first statement in the event handler for the Listen button attaches a method to the `MessageReceived` event that is exposed by a `DatagramSocket`. This event is fired each time a new datagram is received. The second statement binds an "endpoint" to the socket. An endpoint is simply something that systems out on the network can connect to. In this case it

tells the socket to listen on the network address that was entered, and the port that was entered. Note that the `BindEndPointAsync` is asynchronous (the clue is in the name) and may take some time to complete.

Receiving Datagrams

When the receiving socket gets a datagram it must decode and display the information in it. The TCP/IP network is completely unaware of the nature of the data that is being transferred in a datagram. The data itself is a stream of 8 bit values but it might be a word processed document, a spreadsheet or even a video frame from a TV show. It is important that the sender and the receiver agree on the structure of the data that is sent and received. One form of "agreement" is the Hyper Text Transfer Protocol (http) that underpins the World Wide Web.

We are going to use a much simpler agreement so that we can send strings of text from one program to another. The first data element in a datagram will be the number of characters in the string. Because we might want to send more than 255 (the largest number you can hold in an 8 bit value) we are going to send this number as a 32 bit value encoded into four 8 bit values. After the length of the string will follow that number of characters. So, the text "Rob Miles" (which is a great message to send by the way) would be the value 9, followed by the 9 characters that make up my name. The program that wants to send the data must measure the length of the string and send that information before the actual text.

The design of TCP/IP includes checking so that by the time the datagram has arrived it should have been checked for errors. This means that a program reading input can work on the basis that it does not contain any corrupted data. However, because a network connection can fail at any time, there is no guarantee that the message itself will get there, and with the UDP protocol the sender has no idea whether or not the message was received.

```
async void receiveSocket_MessageReceived(DatagramSocket sender,
                    DatagramSocketMessageReceivedEventArgs args)
{
    var dataReader = new DataReader(args.GetDataStream());

    uint numBytesRead;

    while (true)
    {
        numBytesRead = await dataReader.LoadAsync(sizeof(uint));

        if (numBytesRead != sizeof(uint))
            return;

        uint numCharactersInMessage = dataReader.ReadUInt32();

        numBytesRead =
            await dataReader.LoadAsync(numCharactersInMessage);

        if (numBytesRead == 0)
            return;

        string result = dataReader.ReadString(numBytesRead);

        DisplayReceivedMessage(result);

    }
}
```

This is the method that runs when a packet arrives. The arguments to the method can be used to get the `DataStream` for the datagram that has arrived. A `DataStream` is exactly what the name implies, it is a stream of data. On its own it can be hard to work with, but we can use it to create a `DataReader` value which we can ask to read things from the stream for us. Using a `DataReader` is not as simple as just reading from it. That would be too easy. Instead we have to get the `DataReader` to load some bytes from the input stream and then, once it has the bytes in hand, convert the data bytes into the form that we want. In other words, consider the following:

```
numBytesRead = await dataReader.LoadAsync(sizeof(uint));
...
uint numCharactersInMessage = dataReader.ReadUInt32();
```

This is a simplified version of the code in the event handler. The first statement, the call of `LoadAsync` is asked to read a particular number of bytes, in this case the size of an unsigned integer data value (which is actually 4). The second statement makes a call of `ReadUInt32` which fetches those four bytes in the form of an unsigned integer. If we just called `ReadUInt32` first the call would instantly fail, as there would be no data to read. The first method is asynchronous, which means that it can be called with an `await` request.

The `await` keyword is used to flag methods that must be called *asynchronously*. There are two ways you can do things, synchronous and asynchronous. When you do something synchronously what we are really saying is "while you wait". If I have my car serviced "synchronously" it means that I stand in the garage waiting for the "service my car" method to complete, at which point I can take my car and drive home. If I have my car serviced "asynchronously" this means that I go off and do some shopping while the job is being done. At some point the garage will ring me up and deliver a "service complete" event. At which point I go and pick up my car and drive home.

Making asynchronous use of data services is very sensible, particularly on a device such as a Windows Phone. The limitations of the network connection to the phone mean that a network connection may take several seconds to deliver an answer. A program should not be stopped for the time it takes for the server to respond. In previous versions of Windows Phone you had to do lots of extra work setting up the method that would be called when the asynchronous action completed. However, the new `async` and `await` keywords allow us to make asynchronous calls much more easily.

When the compiler sees a method marked with the `await` keyword it generates and calls code that will create a task to run that method and the statements that follow it. This means that developers can use methods that might take a while to complete (for example a request to get data from the network) without having to take any special actions. Lots of the methods in the Windows Phone 8 network code are used this way, as lots of network actions cannot take place instantly. You will also see the await keyword in Chapter 9, when we see that a program can continue running while it is speaking output.

The `receiveSocket_MessageReceived` method repeatedly reads and display strings (length value followed by a number of characters) until there is no more data available. A request to load a number of bytes will return with the number of bytes that were actually found. If this value is less than expected the method returns, as the datagram has been completely read.

The received message is displayed on the screen using a method that takes

```
private void DisplayReceivedMessage(string p)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        ResponseTextBox.Text = p;
    });
}
```

Updating the display from a different thread

You might think display update method looks a little strange. It seems a little over complicated. All it has to do is call the `updateDisplay` method, so you might think it should look like this:

```
private void DisplayReceivedMessage(string p)
{
    ResponseTextBox.Text = p;
}
```

However, this would not work. To understand why it will fail we have to consider how the XAML display is built. Somewhere deep inside the Windows Phone operating system is the part that drives the display. This takes all the elements on the screen and redraws them. To do this it uses the powerful graphics hardware inside the phone, which likes to be driven in a certain way. Essentially the operating system must assemble a bunch of drawing commands and then pass them to the hardware.

This means that the display system must be in total control of the drawing process. If changes are made to display components without the involvement of the display manager the screen may become corrupted, or the draw action fail.

As far as our programs are concerned, the result of this is that only programs running in the "context" of the XAML page are allowed to update display components. Up until now everything we have written has executed as part of the page. For example, the event handler that runs when a button is pressed runs in the display context and can update the screen. However, when the `DatagramSocket` generates events these events are not generated in the context of the display page. Which means that code running from an event produced by the `DatagramSocket` is not allowed to access the display.

We get round this problem by using the Disapatcher object. Every XAML display element has a Dispatcher property associated with it. The Dispatcher exposes a method called BeginInvoke which can be given a task to perform on behalf of a thread which would like access to the display context.

This is exactly the same as me asking someone to climb a ladder and fit a burglar alarm bell because I am afraid of heights (apparently there is a strong correspondence between programmers and fear of heights).

The locater status changed method will call BeginInvoke to perform the display update. If I am sending someone up a ladder I'll give them some instructions as to what to do, for example "Drill three 8mm holes and then bolt the alarm to the wall". In the case of the BeginInvoke method the instructions can be provided by using a "lambda expression".

```
Dispatcher.BeginInvoke(() =>
{
    updateDisplay(args.Position);
});
```

The lambda expression generates a delegate that refers to a block of statements.

```
() =>
{
    ResponseTextBox.Text = p;
}
```

This is what it looks like once it has been removed from the call of BeginInvoke. The item at the top, the () part, means that the code in this lambda expression does not act on any data. If you want to pass parameters into the code inside the lambda expression you can do this, but we don't need to.

The => sequence is the lambda itself, and it is followed by the block of statements that are obeyed. If we were hanging things on the wall this would be where I put the behaviour to drill the holes and fit the alarm. What actually happens is that the lump of code is stashed somewhere and a delegate (which in C# is how we manage pointers to code) is created to pass into the method.

So, when my program runs the BeginInvoke method in the Dispatcher is given a lump of code to run. It adds this lump of code to a list of "things to do" next time it has to update the display. This means that the display doesn't actually update precisely when you ask it to, but the process happens so quickly that this is not a problem.

Creating a Sender

Now that we have our listener, we can create the sender that will send datagrams to it. This will take in what the user types and then send it as a datagram to the destination port. Note that a sender has no way of knowing whether or not the listener actually got the message, unless the listener sends something back in response. It is of course possible for a single system to be both a listener and a sender, this is how we can implement two-way conversations.

To use the sender the user will enter the IP address of the destination device and the port to send the datagram to. They will then enter the text and press the Send button. The string of text that is sent will be encoded so that the listener can understand it, i.e. the sender will send out the number of bytes in the string followed by the string itself.

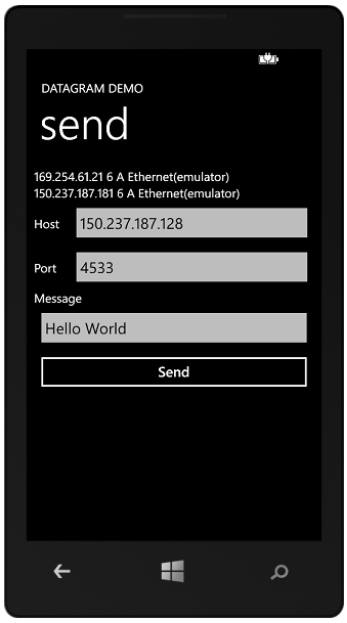


Figure 7-13 The Sender running in the emulator

Figure 7-13 shows the Sender user interface. It uses exactly the same code as the Listener program to show the IP addresses for the phone. In this case the program is running on an emulator and so there are lots of addresses in use.

When the user presses the Send button the event handler for the button takes the text out of the `TextBox` and calls the `SendMessage` method to actually send the message.

```
async private void SendButton_Click(object sender,
                                    RoutedEventArgs e)
{
    await SendMessage(MessageTextBox.Text);
}
```

The `SendMessage` method creates a new `DatagramSocket` and uses it to send the string of text it has been given. You could use this to send any message you like to a distant machine.

```
private async Task SendMessage(string message)
{
    DatagramSocket sendSocket;

    sendSocket = new DatagramSocket();

    await sendSocket.ConnectAsync(new HostName(HostTextBox.Text),
                                  PortTextBox.Text);

    DataWriter dataWriter =
                    new DataWriter(sendSocket.OutputStream);

    dataWriter.WriteUInt32(dataWriter.MeasureString(message));
    dataWriter.WriteString(message);

    await dataWriter.StoreAsync();

    sendSocket.Dispose();
}
```

Note that a `DataWriter` instance is used to send the length of the string, followed by the message. When the message has been sent the program disposes of the socket that was used.

The solutions in *Demo 02a DatagramSend* and *Demo 02b DatagramListen* contain the sender and listener code as described above. It should work on both the emulator and real devices. If you have problems use the IP address at the bottom of the list, and remember that the address you put in the listener is the address of that machine, **not** the remote one.

You might want to think about how you could automate these connections, so that the addresses are entered automatically by the program.

7.5 Creating a Transmission Control Protocol (TCP) Connection

The problem with User Datagram Protocol (UDP) is that systems that use it to transfer data do not know if the recipient received anything. It can also only be used to send one message at a time.

If we want to create reliable, ongoing connections we have to use a different protocol. This is the Transmission Control Protocol we saw earlier. This provides a connection between two machines. You can compare Transmission Control Protocol (TCP) with User Datagram Protocol (UDP) by comparing SMS text messages with phone calls. If I send someone a text message I have no idea whether or not they received it. I will only know that they have got the message when I get a response from them. This is the service provided via UDP. However if I call them on the phone we make a connection. I can send and receive messages during the call and be sure that they are at the other end listening. This the service provided by TCP.

When we use TCP we actually create an on-going connection. Messages are created and transferred in exactly the same way, but each message is part of a call, rather than being an individual event. The call will exist until one of the parties in it decides to terminate the conversation. The response that we receive from sending a message will allow our programs to determine whether the message was delivered successfully or not. At the end of the call one of the parties will close the connection.

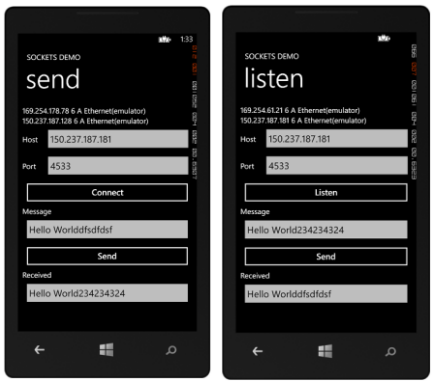


Figure 7-14 Send and listen with Sockets

Figure 7-14 shows how the two programs will work together. The Listen program must be started first and told to listen on a particular port using the IP address of one of the connections on it. The Send program can then run and connect to the listening device. Once the connection has been established either device can send messages to the other. You can see this in action above. The strings of text are encoded using the same technique as for the UDP program earlier, with a length value pre-ceding a string of text characters.

Setting up a Listener

When the user presses the Listen button the listening system creates a `StreamSocketListener` object that will accept incoming connections:

```
StreamSocketListener listenerSocket = new StreamSocketListener();

async private void ListenButton_Click(object sender,
                                      RoutedEventArgs e)
{
    listenerSocket.ConnectionReceived += socket_ConnectionReceived;
    await listenerSocket.BindEndpointAsync(
               new HostName(HostTextBox.Text), PortTextBox.Text);
}
```

This sets up an event handler for the `ConnectionReceived` event and then calls a method to bind the endpoint to the socket. Note that this is an asynchronous method. Note also that it is using exactly the same pattern as the UDP code we saw earlier.

When a client makes a connection to a listening system a `ConnectionReceived` event will be generated and the event handler will run.

```
StreamSocket clientSocket;

void socket_ConnectionReceived(StreamSocketListener sender,
             StreamSocketListenerConnectionReceivedEventArgs args)
{
    clientSocket = args.Socket;
    WaitForMessage();
}
```

This sets up a `StreamSocket` from the arguments supplied to the event and then calls exactly the same `WaitForMessage` method that we used to read the contents of the UDP datagram in the previous example. It repeatedly reads incoming messages and then displays them.

Incoming Addresses

The above event hander just accepts a connection and begins waiting for messages from it. It will accept connections from any remote site.

```
void socket_ConnectionReceived(StreamSocketListener sender,
            StreamSocketListenerConnectionReceivedEventArgs args)
{
    PostReceivedMessageToUI("Connected: " +
                args.Socket.Information.RemoteHostName.DisplayName);
    clientSocket = args.Socket;
    WaitForMessage();
}
```

This version of the connection received event handler displays a message giving the IP address of the remote system that has connected.

Figure 7-15 displaying the connected host

The remote host information is extracted from the arguments to the event. This means that it is possible for a system to only accept request from particular machines.

Setting up a Sender

The sender opens up a TCP connection to the distant listener in a very similar way to the datagram code we saw earlier.

```
async private void ConnectButton_Click(object sender,
                                       RoutedEventArgs e)
{
    clientSocket = new StreamSocket();

    try
    {
        await clientSocket.ConnectAsync(
           new HostName(HostTextBox.Text), PortTextBox.Text);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }

    WaitForMessage();
}
```

The only difference is that this time the connection is made using a `StreamSocket` object.

The send mechanism assembles and sends the message in exactly the same way as for a UDP message, using a DataWriter to send the length followed by the characters.

The solutions in *Demo 03a SocketSend* and *Demo 03b SocketListen* contain the sender and listener code as described above. It should work on both the emulator and real devices. Make sure that you start the listener listening **before** you try to connect using the send program. Note that these programs are not very robust. If one system disconnects the other will throw exceptions as the read actions fail. You might want to consider how these could be caught to make a reliable socket based connection.

7.6 Reading a Web Page

Now that you know about UDP and TCP connections you know all about how the Internet transfers data. Everything that you do with the Internet uses these two methods to move data around. This includes the World Wide Web. In the case of the web the server listens (usually on port number 80) for commands in the Hyper Text Transfer Protocol (HTTP). When it receives a command, for example the word GET, it performs the required action. GET means return a web page.

If you wanted to you could write low level socket based code to set up a TCP connection with a web server and then fetch the data back. However, this is such a common use for programs that the Windows Phone developers have done this for us. They have created a `WebClient` class that uses the internet connection to talk to the server.

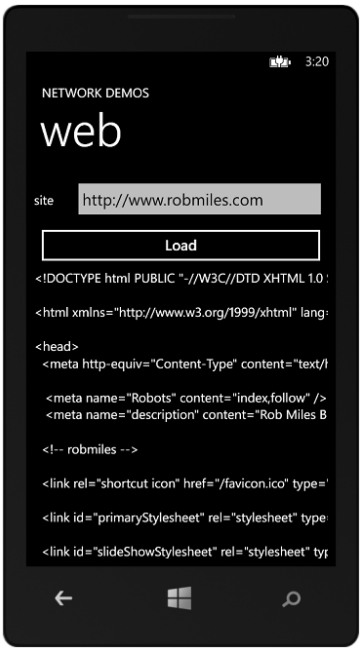


Figure 7-16 Displaying a web page

Figure 7-16 shows the program in action. The user enters a web address, presses the Load button.

When the load button is clicked the following sequence takes place:

1. The text from the `siteTextBox` is used to create a new `Uri` which describes the web site to be loaded.
2. The `DownloadStringAsync` method is called on the `WebClient`. This starts a new web transaction. This method returns immediately, so the button event handler is now completed.
3. Some-time later the `WebClient` will have fetched the string of content at the url. It fires the `DownloadStringCompleted` event to indicate that it has finished, and to deliver the result.
4. The `client_DownloadStringCompleted` method now runs, which checks for errors and displays the received text in the TextBlock.

The code to create the web client and request the page is very simple:

```
private void LoadButton_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();

    client.DownloadStringCompleted +=
                          client_DownloadStringCompleted;
    client.DownloadStringAsync(new Uri(siteTextBox.Text));
}
```

By now you should be used to this pattern. The program binds an event hander that will be used when the action is complete and then fires off a request to perform the download.

The method `client_DownloadStringCompleted` will be executed when the client has fetched the page.

```
void client_DownloadStringCompleted(object sender,
                                DownloadStringCompletedEventArgs e)
{
    if (e.Error != null)
        return;
    receivedTextBlock.Text = e.Result;
}
```

The method just checks to see if an error occurred and if there is no error notification it just puts the result that was received onto the screen.

The solution in *Demo 04 WebScraper* will load and display the text from a web page of your choice.

Note that this is a very basic web page reading program (as you might expect for only fifteen or so lines of code). It does not stop the user from pressing the Load button repeatedly and interrupting downloads and it does not provide any form of time out or progress display to the user.

Adding a Progress Indicator

The program works fine, but a user might not like it because it doesn't keep them informed of what it is doing. Some web pages might take a while to load and it would be nice if the program displayed an indicator to show when it is busy. We can do this by adding a `ProgressIndicator` to the page. This is a XAML component, but rather than add it to the XAML description of the page we are instead going to create the element in software.

```
ProgressIndicator prog;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    prog = new ProgressIndicator();
    prog.IsIndeterminate = true;
    prog.IsVisible = false;
    prog.Text = "Fetching web page";
    SystemTray.SetProgressIndicator(this, prog);
}
```

When the system navigates to the page it will create a `ProgressIndidicator`. The property `IsIndeterminate` is set to true, because with a web page it is impossible to tell how long it will take. You can also use this indicator to show a progress bar along the top of the page. If that is the case you can set this property to false and then update a value in the progress indicator to reflect your programs' progress through the task. Initially, before the user has pressed a button to start

a load, the progress indicator is not visible. The last statement in the method sets the progress indicator for the SystemTray to the one that we have created.

When the program starts a web page loading it sets the `IsVisible` property to true and when the page is displayed the property is set back to false, hiding the indicator.

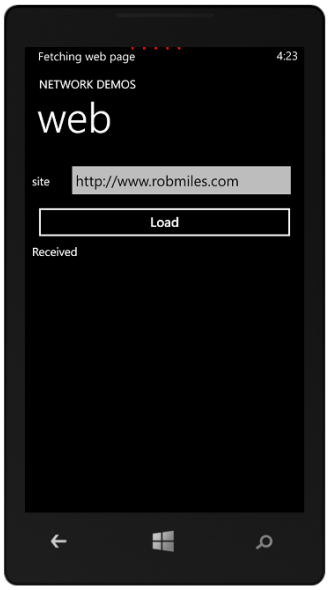


Figure 7-17 The progress indicator

Figure 7-17 shows the progress display in action with the characteristic moving dots along the top of the page.

The solution in *Demo 05 WebScraper Status* will load and display the text from a web page and display a status line when it is busy.

7.7 Using LINQ to Read from an XML Stream

The ability to read from the web can be used for much more than just loading the text part of a web page. We can also interact with many other data services, for example RSS, which stands for "Rather Simple Syndication". This is a format for holding things like blog posts or magazine authors. Lots of sites provide RSS feeds of their content and programs can connect to them and consume their content

As an example, if you point a web browser at the following url:

```
http://www.robmiles.com/journal/rss.xml
```

- you will be rewarded with an XML document that contains my most recent blot posts. Because your browser knows about RSS it will format the XML it receives.

This makes it very easy to make a slightly modified version of our WebClient program that assembles a web request for an RSS feed and use that to read feed information from it.

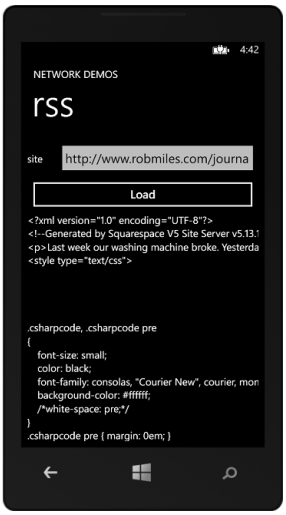


Figure 7-18 Reading an RSS feed

Figure 7-18 shows the actual XML that makes up my blog RSS feed, complete with lament about a broken washing machine. This is the first part of the RSS reader that we are building, at the moment it just displays the text itself. Next we have to convert this into a proper rendering of the data.

We are going to do this by extracting elements from the XML document and then mapping these onto a display template that will be used to build a list of these elements.

Figure 7-19 RSS Feed Display format

Figure 7-19 shows the display format that we are going to use. It is very simple, and just pulls out three elements from each RSS item. The program will show the title, the data published and the url to the actual article. You can use this technique to read any RSS data source, you can even pull out the URLs of images and display those within your templates.

Structured Data and LINQ

An RSS document contains items in the form of an XML formatted document. XML is something we know and love (a bit). What we want to do now is pull all the information out of the XML content we get from the RSS feed and display this information properly. As great programmers we could of course write some software to do this. The good news is that we don't have to. We can use LINQ to do this for us.

An XML document is very like a database, in that contains a number of elements each of which has property information attached to it. Along with tools to create database queries and get structured information from them, LINQ also contains tools that can create objects from XML data. We are going to use these to read our RSS feed.

The RSS feed as a data source

The information that you get in an RSS is a structured document that contains attributes and properties that describe the articles. If you have been paying attention during the discussions of XAML and XML you should find this very familiar.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Generated by Squarespace V5 Site Server v5.13.185 (http://www.squarespace.com) on Mon, 02 Sep
<rss xmlns:content="http://purl.org/rss/1.0/modules/content/" xmlns:wfw="http://wellformedweb.org
  <channel>
    <title>Rob Miles' Journal</title>
    <link>http://www.robmiles.com/journal/</link>
    <description>The Wonderful Life (tm) of Rob Miles</description>
    <lastBuildDate>Wed, 28 Aug 2013 13:00:45 +0000</lastBuildDate>
    <copyright>(c) Rob Miles 2010</copyright>
    <language>en-GB</language>
    <generator>Squarespace V5 Site Server v5.13.185 (http://www.squarespace.com)</generator>
    <item>
      <title>Feminine Logic</title>
      <category>Life</category>
      <dc:creator>Rob</dc:creator>
      <pubDate>Wed, 28 Aug 2013 12:59:36 +0000</pubDate>
      <link>http://www.robmiles.com/journal/2013/8/28/feminine-logic.html</link>
      <guid isPermaLink="false">49484:424394:34208093</guid>
      <description>
        <![CDATA[<p><a title="WP_20130827_17_04_42_Pro.jpg by RobMiles, on Flickr" href="http://w
<p>Last week our washing machine broke. Yesterday I fitted the new one. Only problem is that the
      </description>
      <wfw:commentRss>http://www.robmiles.com/journal/rss-comments-entry-34208093.xml</wfw:commer
    </item>
    <item>
      <title>You really should go to &amp;ldquo;Your Really Should&amp;rdquo; events at C4DI</tit
      <category>C4di</category>
      <category>Windows Phone</category>
      <dc:creator>Rob</dc:creator>
      <pubDate>Tue, 27 Aug 2013 20:17:00 +0000</pubDate>
      <link>http://www.robmiles.com/journal/2013/8/27/you-really-should-go-to-ldquoyour-really-sh
      <guid isPermaLink="false">49484:424394:34207933</guid>
      <description><![CDATA[<a title="WP_20130827_18_14_46_Pro__highres.jpg by RobMiles, on Flick
      <wfw:commentRss>http://www.robmiles.com/journal/rss-comments-entry-34207933.xml</wfw:commer
    </item>
    <item>
      <title>Updating Display Elements in Windows Phone</title>
      <category>Windows Phone</category>
      <dc:creator>Rob</dc:creator>
      <pubDate>Mon, 26 Aug 2013 18:33:00 +0000</pubDate>
```

This is the top half of my RSS document. This is not the complete text, but if you look carefully you can find the text of the blog post itself (look for the lament about our washing machine), along with lots of other stuff. The real document contains a number of items. Above I have only shown a few.

In the same way as we can ask LINQ to query a database and return a collection of objects it has found we can ask LINQ to create a construction that represents the contents of an XML document:

```
XElement rssElements = XElement.Parse(rssText);
```

The `XElement` class is the part of LINQ that represents an XML element. The `rssText` string contains the text that we have loaded from the RSS feed using our `WebClient`. Once this statement has completed we now have an `XElement` that contains all the RSS feed information as a structured document. The next thing we need to do is convert this information into a form that we can display. We are going to pull just the status items that we need out of the `rssElements` and use these to create a collection of items that we can display using data binding.

Creating Posts for Display

We know that it is very easy to bind a C# object to display elements on a XAML page. So now we have to make some objects that we can use in this way.

The items we are going to create will expose properties that our XAML elements will use to display their values. The three things we want to display for each post are the title of the post, the date of the post and a link to the post itself. We can make a class that holds this information.

```
public class RSSPost
{
    public string PostTitle { get; set; }
    public string DatePosted { get; set; }
    public string PostLink { get; set; }
}
```

These properties are all "one way" in that the `RSSPost` object does not want to be informed of changes in them. This means that we can just expose them via properties and all will be well. The `UserImage` property is a string because it will actually give the uri of the image on the internet.

The next thing we have to do is to use LINQ to get an object collection from the `XElement` that was built from the RSS feed we loaded.

```
var postList =
    from item in rssElements.Elements("channel").Elements("item")
        select new RSSPost
        {
            PostTitle = item.Element("title").Value,
            DatePosted = item.Element("pubDate").Value,
            PostLink = item.Element("link").Value
        };
```

This code will do exactly that. It is well worth a close look. The `from` keyword requests an iteration through a collection of elements in the document. The elements are identified by their name. The program will work through each status element in turn in the document. Each time round the loop the variable `tweet` will hold the information from the next status element in the `XElement`.

The `select` keyword identifies what is to be returned from each iteration. For each selected item we want to return a new `RSSPost` instance with settings pulled from that `post`. The `Element` method returns an element with a particular name. As you can see we can call the `Element` method on other elements. That is how we get the `items` out of the `channel` element for the document.

The variable `postList` is set to the result of all this activity. We can make it `var` type, but actually it is a collection of `RSSPost` references. It must be a collection, since the `from` keyword is going to generate an iteration through something.

We now have a collection of `RSSPost` values that we want to put onto the screen.

Laying out the post display using XAML

We have seen how easy it is to put a XAML element on a page and then bind object properties to it. However, we have a little extra complication with our program. We want to display lots of posts, not just one value. Ideally we want to create something that shows the layout for a single item and use that to create lots of display elements, each of which is bound to a different `RSSPost` value. This turns out to be quite easy.

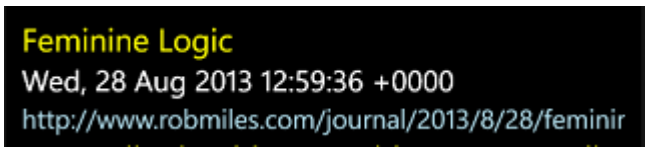


Figure 7-20 RSS Item Display format

Above you can see what I want an individual post to look like. It will have a title at the top, then the date and then the link. I want each to be in different colours and text sizes.

From a XAML point of view this is three elements in a StackPanel. We can get XAML to do a lot of the work in laying these out for us by using the `StackPanel` container element.

```
<StackPanel>
    <TextBlock Text="{Binding PostTitle}" Foreground="Yellow" FontSize="24" />
    <TextBlock Text="{Binding DatePosted}" TextWrapping="Wrap" FontSize="22" />
    <TextBlock Text="{Binding PostLink}" Foreground="LightBlue" FontSize="20" />
</StackPanel>
```

Above you can see the XAML that expresses all this. The `StackPanel` class is a lovely way of laying out items automatically. We don't have to do lots of calculations about where things should be placed, instead we just give the items and the `StackPanel` puts them all in the right place. You can also see the bindings that link the items to the properties in the `RSSPost` class, along with the font size and colour for each item. If we add the complete XAML above to our main page we have an area that can hold a list of RSS items.

Creating a complete layout

```
<ListBox Height="453" Width="456" HorizontalAlignment="Left" Name="RSSListBox"
                                  VerticalAlignment="Top">
    <ListBox.ItemTemplate>
        <DataTemplate>
        <StackPanel>
            <TextBlock Text="{Binding PostTitle}" Foreground="Yellow" FontSize="24" />
            <TextBlock Text="{Binding DatePosted }" TextWrapping="Wrap" FontSize="22" />
            <TextBlock Text="{Binding PostLink}" Foreground="LightBlue" FontSize="20" />
        </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

This is exactly how we do it. This XAML repays a lot of study. The first thing we notice is that this is a `ListBox` element. We've not seem these before but they are, as you might expect, a way of holding a list of boxes. Within a `ListBox` we can put a template that describes what each item in the list will look like. This includes the data binding that will link element properties to our `RSSPost` instances. It also includes the layout elements that will describe how the post will look.

Creating Posts for Display

The next thing our RSS item reader must do is get the item data onto the display. The items have been assembled in to a list, ready for mapping onto the display and Before we do this, it is worth stepping back and considering what we are about to do. This is worth doing because it will help us to understand how XAML binds collections to displays.

We could write some test code that made us a collection of `RSSPost` items:

```
RSSPost p1 = new RSSPost
{
    PostTitle = "title1",
    DatePosted = "date1",
    PostLink = "link1"
};
RSSPost p2 = new RSSPost
{
    PostTitle = "title2",
    DatePosted = "date2",
    PostLink = "link2"
};
RSSPost p3 = new RSSPost
{
    PostTitle = "title3",
    DatePosted = "date3",
    PostLink = "link3"
};

List<RSSPost> posts = new List<RSSPost>();

posts.Add(p1);
posts.Add(p2);
posts.Add(p3);
```

This code creates three `RSSPost` instances and then adds them to a list of posts. This is not particularly interesting code. It becomes interesting when we do this:

```
RSSListBox.ItemsSource = posts;
```

The magic of data binding takes the list of posts and the template above, and does this with it:

Figure 7-21 Displaying sample data

This is very nice. It means that you can display any collection of data in a list on the page just by setting them as the source for a `ListBox`. All you have to do is create a data template that binds the properties of each instance to display elements and then assign a list of these instances to the `ItemSource` property of the `ListBox`.

Since we know that we can assign a collection of `RSSPost` references to a `ListBox` and get this displayed the next statement is not going to come as much of a surprise.

```
RSSListBox.ItemsSource = postList;
```

At this point our work is done, and the RSS posts are displayed.

The `ListBox` provides a lot of useful functionality. If you run this program you will find that you can scroll the list up and down by dragging them with your finger. The scroll action will slow down and the items will even "scrunch" and bounce as they move.

The solution in *Demo 06 RSS Reader* contains this code. You can type in any RSS feed and view the items in it.

This is an extremely powerful feature, it is very easy to get hold of structured data from the web and pull out and format just the elements that you want to display. If you investigate you will find that you can even load the XML data from the web directly into an XML element.

What We Have Learned

1. Computer networks provide a means by which one system can send a message to another. A given system is physically connected to its *local* network. Each local network also contains a *router* component which will send packets off the local network to remote destinations. The routers are connected by data transfer devices which provide *inter-network* connections.
2. The full address of a system on a network contains the address of that station on its physical network along with the address of their physical network on the internet.
3. Data can be transferred between systems on the basis of unacknowledged datagrams (UDP) or as part of a connection set up between two systems (TCP).
4. Individual services on a system can be addressed by the use of ports, some of which have "well known" address values, for example an HTTP web server will typically be located behind port 80 on a host.
5. Windows Phone applications can create and use objects that represent a connection to a network service. The use of these objects is *asynchronous*, in that the results of a request are not returned to that request, but delivered later by a further call made by the network client.
6. A socket provides an abstraction of a connection to a network.
7. Some network calls may return structured data that contains XML describing the content.
8. The LINQ (Language INtegrated Query) library provided for use on the phone can create a representation of structured data which can then be traversed to create structured data for use in a program.
9. LINQ can used be to provide an automatic transfer of data from the rows and tables in a database into objects that can be used in a program.
10. XAML allows the creation of display templates which can be bound to data elements. These templates can include lists, so that a collection of structured data can be bound to a list.

8 XNA Game Development

After all the hard work of the previous sections, now would seem a good place to have some fun and play some games. You can write games using XAML, but that is not really what it was designed for. XNA on the other hand was built from the ground up to be an efficient and powerful tool for game creation. In this section we are going to take a look at XNA and how to use it to create games for the Windows Phone device.

Microsoft developed XNA up to Version 4.0, which allowed developers to create games for Windows 7 PC, Xbox 360 and Windows Phone up to version 7.5. However, at that point they suspended development of the framework, although it is still present in Visual Studio 2012.

However, a team of Open Source developers have taken a shine to XNA and have been porting the framework over to many other platforms including iOS and Android. They have also produced a version of MonoGame for Windows 8 and Windows Phone 8. Converting from XNA 4.0 to MonoGame is not difficult, and we will investigate this at the end of the chapter.

8.1 XNA in context

XNA is for creating games. It provides a complete ecosystem for game creation, including Content Management (how you get your sound effects, maps and textures into a game) and the game build process (how you combine all the elements into the single distributable game element). We are not going to delve too deeply into all these aspects of the system, instead we are going to focus on the XNA Framework, which is a library of C# objects that is used to create the games programs themselves.

In this chapter we are going to write XNA 4.0 games for Windows Phone 7.5. These will also run on Windows Phone 8 devices. The disadvantage of XNA 4.0 games is that they are not able to use more advanced Windows Phone 8 features, for example voice response. The advantage for us, from a learning point of view, is that XNA 4.0 games can be run in the Windows Phone emulator, which is not possible with the current version of MonoGame.

2D and 3D Games

Games can be “2D” (flat images that are drawn in a single plane) or “3D” (a visual simulation of a 3D environment). XNA provides support for both kinds of games and the Windows Phone hardware acceleration makes it capable of displaying realistic 3D worlds. For the purpose of this section we are going to focus on 2D, sprite based, games however. We can create a good gameplay experience this way, particularly on a mobile device.

8.2 Making an XNA 4.0 program

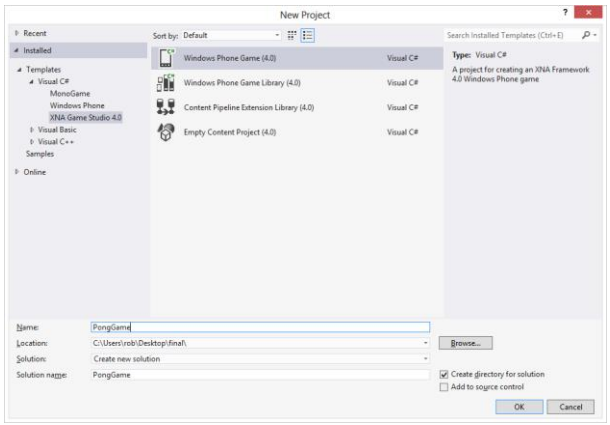


Figure 8-1 Creating an XNA 4.0 game

The New Project dialog in Visual Studio lets us select Windows Phone Game 4.0 as the project type.

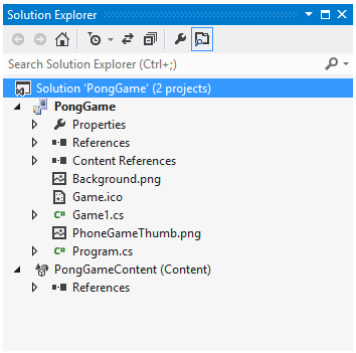


Figure 8-2 An XNA Game solution

The Solution for an XNA project looks rather like a XAML one, but there are some differences. There is a second project in the solution which is specifically for game content. This is managed by the Content Manager which provides a set of input filters for different types of resources, for example PNG images, WAV sound files etc. These resources are stored within the project and deployed as part of the game onto the target platform. When the game runs the Content Manager loads these resources for use in the game. The result of this is that whatever the platform you are targeting the way that content assets are managed is the same as far as you are concerned.

If you run the new game project as created above you will be rewarded by a blue screen. This is not the harbinger of doom that it used to be, but simply means that the initial behaviour of an XNA game is to draw the screen blue.

Figure 8-3 An empty game

Note that by default the game runs in Landscape mode, and that part of the screen is taken for the display of the phone status information. We will find out how to disable this display, and also use the phone in different orientations, later in the chapter.

How an XNA Game Runs

If you look at a XAML application you will find that a lot of the time the program never seems to be doing anything. Actions are only carried out in response to events, for example when a user presses a button, or a network transaction completes, or a new display page is loaded.

XNA programs are quite different. An XNA program is continuously active, updating the game world and drawing the display. Rather than waiting for an input from a device, an XNA game will check devices that may have input and use these to update the game model and then redraw the screen as quickly as possible. In a game context this makes a lot of sense. In a driving game your car will continue moving whether you steer it or not (although this might not end well). The design of XNA recognises that the game world will update all the time around the player, rather than the player initiating actions.

When an XNA game runs it actually does three things:

1. Load all the content required by the game. This includes all the sounds, textures and models that are needed to create the game environment.
2. Repeatedly run the Game Engine:

- Update the game world. Read controller inputs, update the state and position of game elements.
- Draw the game world. Take the information in the game world and use this to render a display of some kind, which may be 2D or 3D depending on the game type.

These three behaviours are mapped onto three methods in the Game class that is created by the MonoGame new project wizard as part of a new game project.

```
partial class Game1 : Microsoft.Xna.Framework.Game
{

    protected override void LoadContent
                              (bool loadAllContent)
    {
    }
    protected override void Update(GameTime gameTime)
    {
    }
    protected override void Draw(GameTime gameTime)
    {
    }
}
```

When we write a game we just have to fill in the content of these methods. Note that we never actually call these methods ourselves, they are called by the XNA framework when the game runs. `LoadContent` is called when the game starts. The `Draw` method is called as frequently as possible and `Update` is called thirty times a second on a Windows Phone device. Note that this is not the same as XNA games on the Windows PC. In order to reduce power consumption a Windows Phone game only updates 30 times a second, rather than the 60 times a second of the desktop version of the game. If you want to write a game which will work on multiple platforms you will have to bear this in mind.

Before we can start filling in the `Update` and `Draw` methods in our game we need to have something to display on the screen. We are going to start by drawing a white ball. Later we will start to make it bounce around the screen and investigate how we can use this to create a simple bat and ball game.

Game Content

We use the word *content* to refer to all the assets that make a game interesting. This includes all the images on the textures in a game, the sound effects and 3D models. The XNA framework Content Management system can take an item of content from its original source file (perhaps a PNG image) all the way into the memory of the game running on the target device. This is often called a *content pipeline* in that raw resources go into one end and they are then processed appropriately and finally end up in the game itself.

XNA Content and MonoGame

Managing content in Microsoft XNA is very easy. If you create a Microsoft XNA project all the content is placed in a separate project which is part of the game solution. The XNA content management system opens content items that we add to the project. They are converted into a form that can be used directly by the game when it runs.

Content management in MonoGame is a little more difficult, because

The content management is integrated into Visual Studio. Items of content are managed in the same way as files of program code. We can add them to a project and then browse them and manage their properties.

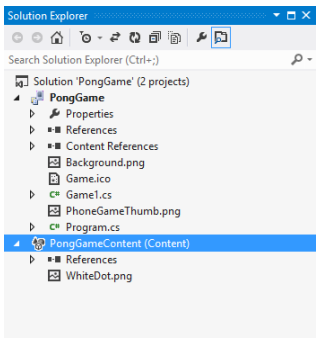


Figure 8-4 The Content Project with an item of content

Above you can see an item of content which has been added to a Content project. The item is a PNG (Portable Network Graphics) image file that contains a White Dot which we could use as a ball in our game. This will have the asset name “WhiteDot” within our XNA game. Once we have our asset as part of the content of the game we can use it in games.

We have seen something that looks a bit like this in the past when we added resources to applications. Those resources have actually been part of Visual Studio solutions as well. However, it is important that you remember that the Content Manager is specifically for XNA games and is how you should manage XNA game content.

Loading Content

The `LoadContent` method is called to load the content into the game. (The clue is in the name). It is called once when the game starts running.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to
    // draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

}
```

The first thing that `LoadContent` does is actually nothing to do with content at all. It creates a new `SpriteBatch` instance which will be used to draw items on the screen. We will see what the `SpriteBatch` is used for later.

We can add a line to this method that loads image into our game:

```
ballTexture  = Content.Load<Texture2D>("WhiteDot");
```

The `Load` method provided by the content manager is given the type of resource to be loaded (in this case `Texture2D`). The appropriate loader method is called which loads the texture into the game.

```
Texture2D ballTexture;
```

XNA provides a type called `Texture2D` which can hold a single two-dimensional texture in a game. Later in the program we will draw this on the screen. A game may contain many textures. At the moment we are just going to use one.

Creating XNA Sprites

In computer gaming terms a sprite is an image that can be positioned and drawn on the display. In XNA terms we can make a sprite from a texture (which gives the picture to be drawn) and a rectangle (which determines the position on the screen). We already have the texture, now we need to create the position information.

XNA uses a coordinate system that puts the origin (0,0) of any drawing operations in the top left hand corner of the screen. This is not same as a conventional graph, where we would expect the origin to be the bottom left hand corner of the graph. In other words, in an XNA game if you increase the value of Y for an object this causes the object to move down the screen.

The units used equate to pixels on the screen itself. Most Windows Phones have a screen resolution of 800 x 480 pixels. If we try to draw objects outside this area XNA will not complain, but it won't draw anything either.

XNA provides a `Rectangle` structure to define the position and size of an area on the screen.

```
Rectangle ballRectangle;
```

This variable can be used to keep track of the position and size of the ball on the screen.

```
ballRectangle = new Rectangle(
        0, 0,
        ballTexture.Width, ballTexture.Height);
```

The above code creates a rectangle which is positioned at the top left hand corner of the screen. It is the same width and height as the ball texture.

Drawing Objects

Now that we have a sprite the next thing we can do is make the game draw this on the screen. To do this we have to fill in the `Draw` method.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    base.Draw(gameTime);
}
```

This is the "empty" `Draw` method that is provided by Visual Studio when it creates a new game project. It just clears the screen to blue. We are going to add some code to the method that will draw our ball on the screen. Before we can do this we have to understand a little about how graphics hardware works.

Modern devices have specialised graphics hardware that does all the drawing of the screen. A Windows Phone has a Graphics Processor Unit (GPU) which is given the textures to be drawn, along with their positions, and then renders these for the game player to see. When a program wants to draw something it must send a batch of drawing instructions to the GPU. For maximum efficiency it is best if the draw requests are be batched together so that they can be sent to the GPU in a single transaction. We do this by using the `SpriteBatch` class to do the batching for us.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    // Draw operations go here

    spriteBatch.End();

    base.Draw(gameTime);
}
```

When we come to write our game the only thing we have to remember is that our code must start and end a batch of drawing operations, otherwise the drawing will not work.

We now have our texture to draw and we know how the draw process will work, the next thing to do is position the draw operation on the screen in the correct place. Once we have done this we will have created a *sprite*. The `SpriteBatch` command that we use to draw on the screen is the `Draw` method:

```
spriteBatch.Draw(ballTexture, ballRectangle, Color.White);
```

The `Draw` method is given three parameters; the texture to draw, a `Rectangle` value that gives the draw position and the colour of the "light" to shine on the texture when it is drawn. The complete `Draw` method looks like this

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    spriteBatch.Draw(ballTexture, ballRectangle, Color.White);

    spriteBatch.End();

    base.Draw(gameTime);
}
```

When we run this program we find that the ball is drawn in the top left hand corner of the screen on the phone:

Figure 8-5 Drawing a white dot

Note that, by default, XNA games expect the player to hold the phone horizontally, in "landscape" mode, with the screen towards the left. We will see later that your game can change this arrangement if it needs to.

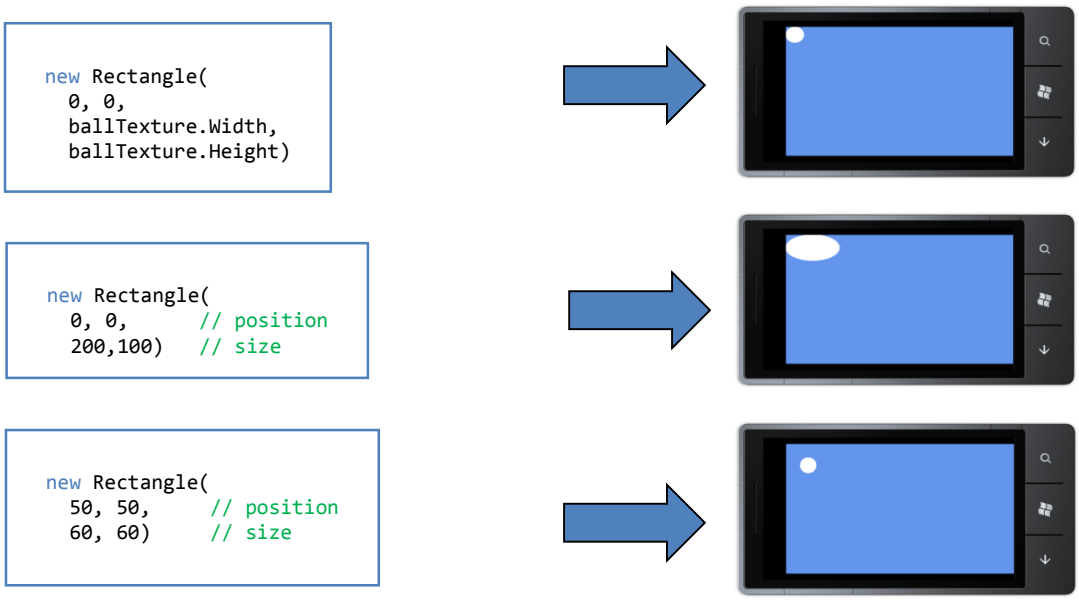


Figure 8-6 Distorting a drawing

The first version of our program drew the ball with the same size as the original texture file. However, we can use whatever size and position we like for the drawing process and XNA will scale and move the drawing appropriately, as you can see above.

The solution in *Demo 01White Dot* contains a Windows Phone XNA game that just draws the white dot in the top left hand corner of the display.

Screen Sizes and Scaling

When you make a game it is important that it looks the same whenever it is played, irrespective of the size of the screen on the device used to play it. Windows Phone devices have a particular set of resolutions available to them. If we want to make a game that looks the same on each device the game must be able to determine the dimensions of the screen in use and then scale the display items appropriately.

An XNA game can obtain the size of the screen from the properties of the viewport used by the graphics adapter.

```
ballRectangle = new Rectangle(
    0, 0,
    GraphicsDevice.Viewport.Width / 20,
    GraphicsDevice.Viewport.Width / 20);
```

This code creates a ball rectangle that will be a 20th of the width of the display, irrespective of the size of the screen that is available.

Updating Gameplay

At the moment the game draws the ball in the same place every time. A game gives movement to the objects in it by changing their position each time they are drawn. The position of objects can be updated in the aptly named `Update` method. This is called thirty times a second when the game is running. We could add some code to update the position of the ball:

```
protected override void Update(GameTime gameTime)
{

    ballRectangle.X++;
    ballRectangle.Y++;

    base.Update(gameTime);
}
```

This version of `Update` just makes the X and Y positions of the ball rectangle one pixel bigger each time it is called. This causes the ball to move down (remember that the Y origin is the top of the screen) and across the screen at a rate of one pixel every thirtieth of a second. After fifteen seconds or so the ball has left the screen and will continue moving (although not visible) until we get bored and stop the game from running.

The solution in *Demo 02Moving Dot* contains a Windows Phone XNA game that draws a white dot that slowly moves off the display.

Using floating point positions

The code above moves the ball one pixel each time `Update` is called. This doesn't give a game very precise control over the movement of objects. We might want a way to move the ball very slowly. We do this by creating floating point variables that will hold the ball position:

```
float ballX;
float ballY;
```

We can update these very precisely and then convert them into pixel coordinates when we position the draw rectangle:

```
ballRectangle.X = (int)(ballX + 0.5f);
ballRectangle.Y = (int)(ballY + 0.5f);
```

The above statements take the floating point values and convert them into the nearest integers, rounding up if required. We can also use floating point values for the speed of the ball:

```
float ballXSpeed = 3;
float ballYSpeed = 3;
```

Each time the ball position is updated we now apply the speed values to the ball position:

```
ballX = ballX + ballXSpeed;
ballY = ballY + ballYSpeed;
```

Now that we can position the ball and control the speed very precisely we can think about making it bounce off the edge of the screen.

Making the ball bounce

Our first game just made the ball fly off the screen. If we are creating some kind of bat and ball game we need to make the ball "bounce" when it reaches the screen edges. To do this the game must detect when the ball reaches the edge of the display and update the direction of movement appropriately. If the ball is going off the screen in a horizontal direction the game must reverse the X component of the ball speed. If the ball is going of the top or the bottom of the screen the game must reverse the Y component of the ball speed.

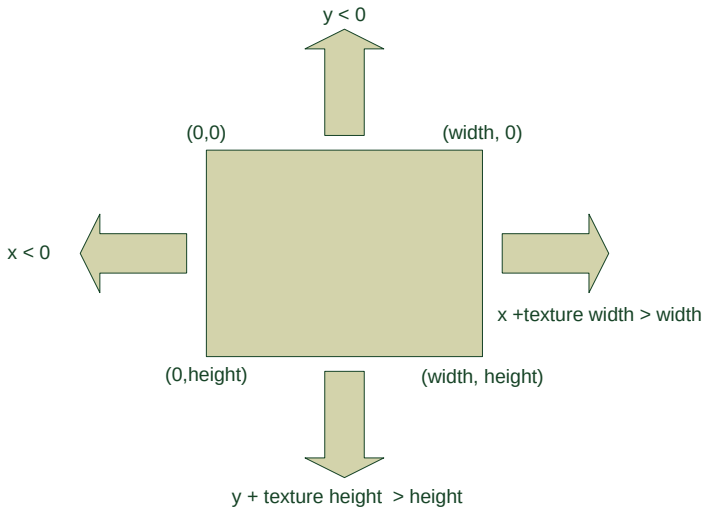


Figure 8-7 Detecting off the screen

Figure 8-7 above shows the directions the ball can move off the screen and the conditions that become true when the ball moves in that direction.

```
if (ballX < 0 ||
    ballX + ballRectangle.Width > GraphicsDevice.Viewport.Width)
{
    ballXSpeed = -ballXSpeed;
}
```

The code above reverses the direction of the ball in the X axis when the ball moves off either the left or right hand edge of the display. We can use a similar arrangement to deal with movement in the Y direction.

The solution in *Demo 03 Bouncing Ball* contains a Windows Phone XNA game that draws a ball that bounces around the display.

Adding Paddles to Make a Bat and Ball Game

We now have a ball that will happily bounce around the screen. The next thing that we need is something to hit the ball with.

Figure 8-8 Paddle design

The paddle is a rectangular texture which is loaded and drawn in exactly the same way as the ball. The finished game uses two paddles, one for the left hand player and one for the right. In the first version of the game we are going to control the left paddle and the computer will control the right hand one. This means that we have three sprites on the screen. Each will need a texture variable to hold the image on the sprite and a rectangle value to hold the position of the sprite on the screen.

```
// Game World
Texture2D ballTexture;
Rectangle ballRectangle;
float ballX;
float ballY;
float ballXSpeed = 3;
float ballYSpeed = 3;


Texture2D lPaddleTexture;
Rectangle lPaddleRectangle;
float lPaddleSpeed = 4;
float lPaddleY;

Texture2D rPaddleTexture;
Rectangle rPaddleRectangle;
float rPaddleSpeed = 4;
float rPaddleY;

// Distance of paddles from screen edge
int margin;
```

These variables represent our "Game World". The `Update` method will update the variables and the `Draw` method will use their values to draw the paddles and ball on the screen in the correct position.

When the game starts running the `LoadContent` will fetch the images using the Content Manager and then set up the draw rectangles for the items on the screen.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which is used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    ballTexture = Content.Load<Texture2D>("ball");
    lPaddleTexture = Content.Load<Texture2D>("lpaddle");
    rPaddleTexture = Content.Load<Texture2D>("rpaddle");

    ballRectangle = new Rectangle(
        GraphicsDevice.Viewport.Width/2,
        GraphicsDevice.Viewport.Height/2,
        GraphicsDevice.Viewport.Width / 20,
        GraphicsDevice.Viewport.Width / 20);

    margin = GraphicsDevice.Viewport.Width / 20;

    lPaddleRectangle = new Rectangle(
        margin, 0,
        GraphicsDevice.Viewport.Width / 20,
        GraphicsDevice.Viewport.Height / 5);

    rPaddleRectangle = new Rectangle(
        GraphicsDevice.Viewport.Width -
            lPaddleRectangle.Width - margin, 0,
        GraphicsDevice.Viewport.Width / 20,
        GraphicsDevice.Viewport.Height / 5);

    lPaddleY = lPaddleRectangle.Y;
    rPaddleY = rPaddleRectangle.Y;
}
```

This `LoadContent` method repays careful study. It loads all the textures and then positions the paddle draw positions each side of the screen as shown below.

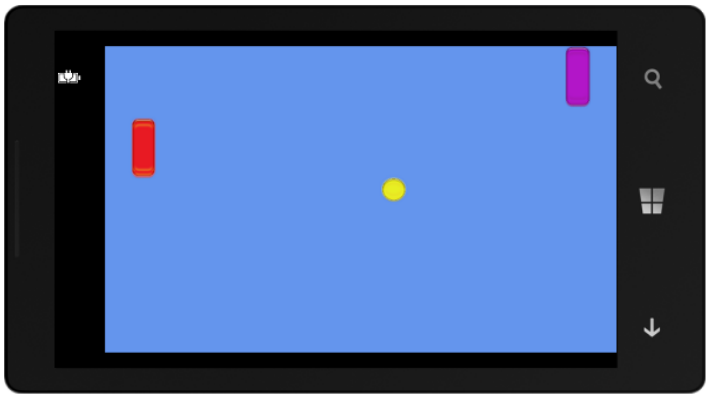


Figure 8-9 Pong

When the game starts the ball will move and the paddles will be controlled by the player.

Controlling Paddles Using the Touch Screen

We can use the Windows Phone touch screen to control the movement of a paddle. We can actually get very precise control of items on the screen, but we are going to start very simple. If the player touches the top half of the screen their paddle will move up. If they touch the bottom half of the screen their paddle will move down.

The Touch Panel returns a collection of touch locations for our program to use:

```
TouchCollection touches = TouchPanel.GetState();
```

Each `TouchLocation` value contains a number of properties that tell the game information about the location, including the location on the screen where the touch took place. We can use the Y component of this location to control the movement of a paddle.

```
if (touches.Count > 0)
{
    if (touches[0].Position.Y > GraphicsDevice.Viewport.Height / 2)
    {
        lPaddleY = lPaddleY + lPaddleSpeed;
    }
    else
    {
        lPaddleY = lPaddleY - lPaddleSpeed;
    }
}
```

The first statement in this block of code checks to see if there are any touch location values available. If there are it gets the location of the first touch location and checks to see if it is above or below the mid-point of the screen. It then updates the position of the paddle accordingly. If we run this program we find that we can control the position of the left hand paddle.

The solution in *Demo 04 Paddle Control* contains a Windows Phone XNA game that draws a ball that bounces around the display. You can also control the movement of the left hand paddle by touching the top or bottom of the screen area.

You can use the touch panel for much more advanced purposes than the example above. You can track the location and movement of a particular touch event very easily. You can also ask the touch panel to detect gestures that you are interested in.

Detecting Collisions

At the moment the bat and the ball are not “aware” of each other. The ball will pass straight through the bat rather than bounce off it. What we need is a way to determine when two rectangles intersect.

The `Rectangle` structure provides a method called `Intersects` that can do this for us:

```
if (ballRectangle.Intersects(lPaddleRectangle))
{
    ballXSpeed = -ballXSpeed;
}
```

The above code tests to see if the ball has hit the left hand paddle. If it has the direction of movement of the ball is reversed to make the ball bounce off the paddle.

Keeping score

For the game to be worth playing it must also keep a score and display this for the players. Points are scored by hitting the back wall of the opponent player. We already have the code that does this. At the moment it just reverses the direction of movement of the ball, it is a simple matter to add a score variable for each player and increase this each time a "goal" is scored. The final thing we need to be able to do at this point is display the score for the players to see. To do this we need to find out how an XNA game draws text for a player.

Displaying Text

The XNA environment does not provide direct access to text drawing. To write text on the screen a game must load a character font and then use this to draw the characters. The font to be used is an item of game content which we need to add to the game. When the font is added the size and the font design is selected. When the game is built the Content Manager will build a set of character images and add these to the game. The game can then draw the text on the game screen along with the other game items.

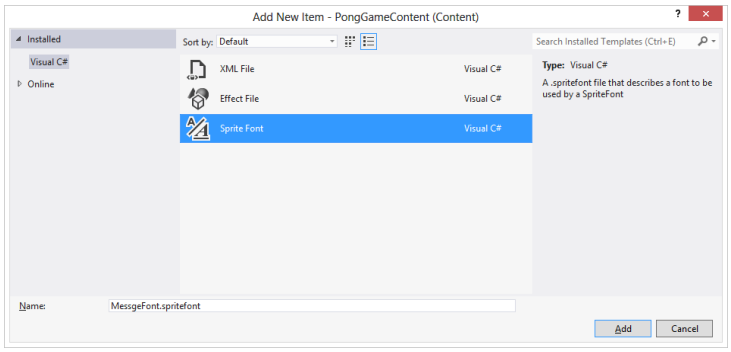


Figure 8-10 Creating a SpriteFont to display messages

The font design, called a "SpriteFont", is created as a new item and given a resource name, in this case MessageFont. The actual font information is held in an XML file which you can edit with Visual Studio:

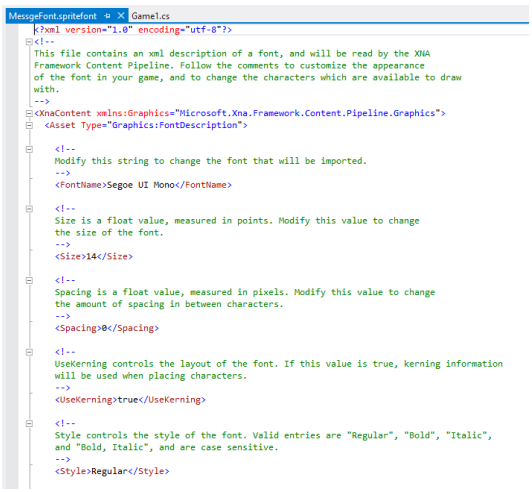


Figure 8-11 Creating a SpriteFont to display messages

This file is where the font size and the name of the font are selected. Above you can see that the font "Segoe UI Mono" is being used with a font size of 14. Further down the XML file you will find settings for bold and italic versions of the font.

Once the `SpriteFont` has been added to the content in a program it can then be loaded when the program runs:

```
SpriteFont font;

font = Content.Load<SpriteFont>("MessageFont");
```

Note that this is exactly the same format as the Load we used to fetch the textures in our game only this time it is being used to fetch a `SpriteFont` rather than a `Texture2D`.

Once we have the font we can use it to draw text:

```
spriteBatch.DrawString( font, "Hello", new Vector2(50,100),
    Color.White);
```

The `DrawString` method is given four parameters. These are the font to use, the string to display, a vector to position the text on the screen and the colour of text to draw. A vector is a way that we can specify a position on the screen, in the example code above the text would be drawn 50 pixels across the screen and 100 pixels down (remember that the origin for Y is at the top of the screen).

The solution in *Demo 05 Complete Pong Game* contains a Windows Phone XNA game that implements a working pong game. The left hand paddle is controlled by the touch panel. The right hand paddle is controlled by an ultra-intelligent AI system that makes it completely unbeatable. Take a look at the code to discover how this top secret technology works. This game is not perfect. Because of the way the ball movement is managed it can sometimes get "stuck" on a paddle or off the edge of the screen. It is up to you to make a completed version of this.

8.3 Player interaction in games

The touch panel is not the only novel input device provided by the Windows Phone. Games can also make use of the accelerometer so that we can create games that are controlled by the tipping of the phone. The accelerometer can measure acceleration in three axes. While it can be used to measure acceleration (you could use your Windows Phone to measure the acceleration of your sports car if you like) it is most often used to measure the orientation of the phone. This is because the accelerometer is acted on by gravity, which gives a constant acceleration of 1 towards the centre of the earth.

You can visualize the accelerometer as a weight on the end of a spring, attached to the back of the phone. The picture below shows us how this might look.

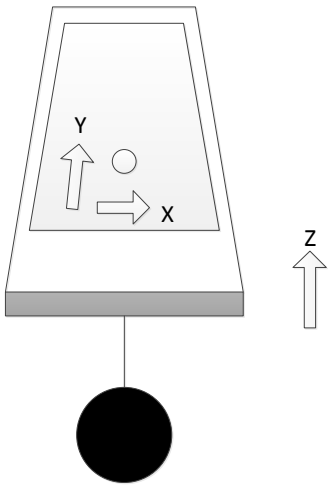


Figure 8-12 Visualizing a accelerometer

If we hold the phone flat as shown in Figure 8-12, the weight will hang straight down underneath the phone. If we were to measure the distance in the X, Y and Z directions of the weight relative to the point where it is attached to the phone it would read 0, 0, -1, assuming that the spring is length 1. The value of Z is -1 because at the moment the position of the weight is below the phone and the coordinate system being used has Z, the third dimension, increasing as we move up from the display.

If you tip the bottom of the phone up so that the far edge of the phone is now pointing towards your shoes the weight would swing away from you, increasing the value of Y that it has relative to the point where the string is attached. If you tip the bottom of the phone down, so that the phone tilts towards you and you can see the screen properly, the weight moves the other way, and the value of Y becomes less than 0. If the phone is vertical (looking a bit like a tombstone) the weight is directly below and in line with it. In this situation the value of Z will be 0, and the value of Y will be -1. Twisting the phone will make the weight move left or right, and will cause the value of X to change.

These are the values that we actually get from the accelerometer in the phone itself. So, at the moment the accelerometer seems to be measuring orientation (i.e. the way the phone is being held) not acceleration. If the phone starts to accelerate in other directions you can expect to see the values in the appropriate direction change as well. Our diagram above actually

helps us visualise this too. If we push the phone away from us the weight will “lag” behind the phone for a second until it caught up resulting in a brief change in the Y value. We would see exactly the same effect in the readings from the accelerometer if we did move the phone in this way.

Getting Readings from the Accelerometer Class

When we used the touch panel from XNA we found that we just had to ask the panel for a list of active touch locations. It would be nice if we could do the same with the accelerometer but this is not how it works. The accelerometer driver has been written in a way that makes it useable in Windows Phone applications as well as games. In XAML pages we were used to receiving messages from things when we wanted them to tell us something. The elements on a XAML page generate events when they are used and web request cause an event when they have data for us to use.

The accelerometer in Windows Phone works the same way. A program must make an instance of the `Accelerometer` class and connect a method to the `ReadingChanged` event that the class provides. Whenever the hardware has a new acceleration reading it will fire the event and deliver some new values for our program to use. If the program is written using XAML it can use those values directly and perhaps bind them to properties of elements on the screen. If the program is written using XNA it must make a local copy of the new values so that they used by code in the `Update` method next time it is called.

Using the Accelerometer Values

Before we can use the `Accelerometer` class we need to add the system library that contains it.

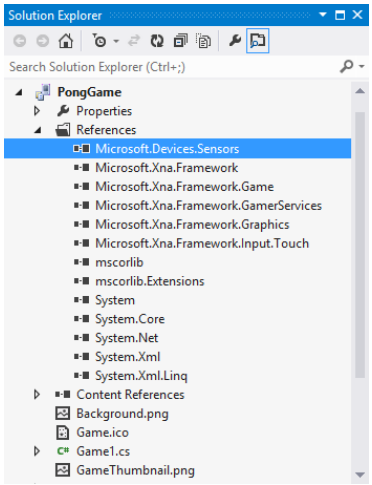


Figure 8-13 Adding the Sensors library

We can also add the appropriate namespace:

```
using Microsoft.Devices.Sensors;
```

Now the game can create an instance of the accelerometer, connect an event handler to it and then start the accelerometer running.

```
protected override void Initialize()
{
    Accelerometer acc = new Accelerometer();
    acc.ReadingChanged +=
            new EventHandler<AccelerometerReadingEventArgs>
                                            (acc_ReadingChanged);
    acc.Start();
    base.Initialize();
}
```

We can do this work in the `Initialise` method, as shown above. This is another method provided by XNA. It is called when a game starts running. It is a good place to put code to set up elements in our program. Next we have to create our event handler. This will run each time the accelerometer has a new value for the game.

```
Vector3 accelState = Vector3.Zero;

void acc_ReadingChanged
                (object sender, AccelerometerReadingEventArgs e)
{
    accelState.X = (float)e.X;
    accelState.Y = (float)e.Y;
    accelState.Z = (float)e.Z;
}
```

This method just copies the readings from the arguments to the method call into a `Vector3` value. The `Update` method in our game can then use these values to control the movement of the game paddle:

```
lPaddleY = lPaddleY - (accelState.X * lPaddleSpeed);
```

This code is very simple. It uses the X value of the accelerometer state to control the movement up and down of the left hand side paddle. You might thing we should use the Y value from the accelerometer, but remember that our game is being played in *landscape* mode (with the phone held on its side) and the accelerometer values are always given as if the phone is held in *portrait* mode.

The accelerometer, in association with a very simple physics model, can be used to create "tipping" games, where the player has to guide a ball around a maze or past obstacles.

The solution in *Demo 06 Tipping Pong* contains a Windows Phone XNA game that implements a working pong game. The left hand paddle is controlled by tipping the phone to make the paddle move. You will notice that the further you tip the phone the faster the paddle moves, which is just how it should be.

Using the Accelerometer Emulation

If you start the program in the emulator you can use the Additional Tools menu to allow you to test the behaviour of the game by simulating the tipping of the emulated phone.

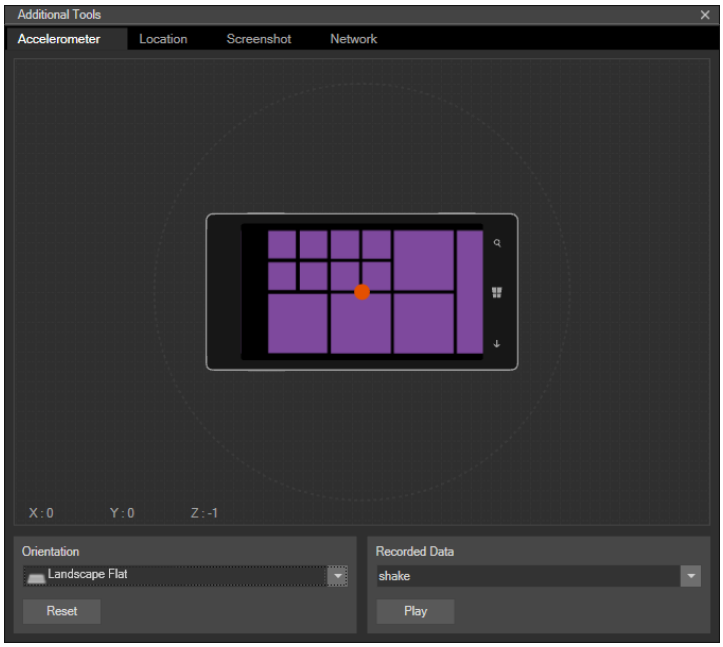


Figure 8-14 Accelerometer Emulation

If click the >> on menu tab just to the right of the emulator display on your screen you can open the Advanced Tools dialog. By selecting the Accelerometer tab and the Landscape Flat orientation you can get a 3D display of the phone which you can tip by dragging the red dot around. This allows you to get a feel of how the accelerometer input works. This is not a true guide to gameplay, but it does allow you to check that the game is using the correct axes.

Threads and Contention

The version of the accelerometer code above will work OK, but code like this can be vulnerable to a problem because of the way the program is written. In the case of this version of the game it would not drastically affect gameplay, but this

issue is worth exploring because you may fall foul of similar problems when you start writing programs like these. Mistakes like these can give rise to the worst kind of programming bug imaginable, which is where the program works fine 99.999 per cent of the time but fails every now and then.

I hate bugs like these. I'm much happier when a program fails completely every time I run it because I can easily dive in and start looking for problems. If the program only fails once in a blue moon I have to wait around until I see the fault occur.

The problem has to do with the way that accelerometer readings are created and stored. We have two processes running working at the same time.

- The accelerometer is generating readings and storing them
- The Update method is using these readings to move the paddle

However, the Windows Phone only has one computer in it, which means that in reality only one of these processes can ever be active at any given time. The operating system in Windows Phone gives the illusion of multiple computer processors by switching rapidly between each active process. This is the cause of our problem. Consider the following sequence of events:

1. Update runs and reads the X value of the acceleration
2. The Accelerometer event fires and starts running. It generates and stores new values of X, Y and Z
3. Update reads the Y and Z values from the updated values
4. Update now has "scrambled" data made up of a mix of old and new readings.

In the case of our game we don't have much of a problem here, in that it only uses the X value of the accelerometer anyway. But if we had a more advanced game which used all three values the result would be that every now and then the `Update` method would be given information that wasn't correct. In an even larger and more complex program that used multiple processes this could cause huge problems.

Of course the problem will only happen every now and then, when the timing of the two events was very close together, but the longer the program runs the more chance there is of the problem arising.

The way to solve the problem is to recognise that there are some operations in a program that should not be interruptible. We need a way to stop the accelerometer process from being able to interrupt the `Update` process, and vice versa. The C# language provides a way of doing this. It is called a `lock`.

A given process can grab a particular lock object and start doing things. While that process has the lock object it is not possible for another process to grab that object. In the program we create a lock object which can be claimed by either the event handler or the code in `Update` that uses the accelerometer value:

```
object accelLock = new object();

void acc_ReadingChanged
                 (object sender, AccelerometerReadingEventArgs e)
{
    lock (accelLock)
    {
        accelState.X = (float)e.X;
        accelState.Y = (float)e.Y;
        accelState.Z = (float)e.Z;
    }
}
```

The variable `accelLock` is of the simplest possible type, that of object. We are not actually storing any data in this object at all. Instead we are just using it as a token which can be held by one process or another. This is the new code in `Update` that uses the accelerometer reading and is also controlled by the same lock object

```
lock (accelLock)
{
    lPaddleY = lPaddleY - (accelState.X * lPaddleSpeed);
}
```

When a process tries to enter a block of code protected by a `lock` it will try to grab hold of the lock object. If the object is not available the process will be made to wait for the lock to be released. This means that it is now not possible for one method to interrupt another. What will happen instead is that the first process to arrive at the block will get the lock object and the second process will have to wait until the lock is released before it can continue.

Locks provide a way of making sure that processes do not end up fighting over data. It is important that any code protected by a lock will complete quickly, so that other processes do not have to spend a lot of time waiting for access to the lock.

It is also important that we avoid what is called the “Deadly Embrace” problem where process A has obtained lock X and is waiting for lock Y, and process B has obtained lock Y and is waiting for lock X. We can help prevent this by making a process either a “producer” which generates data or a “consumer” which uses data. If no processes are both consumers and producers it is unlikely that they will be stuck waiting for each other.

8.4 Adding sound to a game

We can make games much more interesting by adding sound effects to them. As far as XNA is concerned a sound in a game is just another form of game resource. There are two kinds of sounds in a game. There are sound effects which will accompany particular game events, for example the sound of a spaceship exploding when it is hit with a missile, and there is background music which plays underneath the gameplay.

Sound effects are loaded into the game as content. They start as WAV files and are stored in memory when the game runs. They are played instantly on request. Music files can also be supplied as game content but are played by the media player.

Creating Sounds

There are many programs that can be used to prepare sound files for inclusion in games. A good one is the program called Audacity. This provides sound capture and editing facilities. It can also convert sound files between different formats and re-sample sounds to reduce the size of sound files. Audacity is a free download from *http://audacity.sourceforge.net*

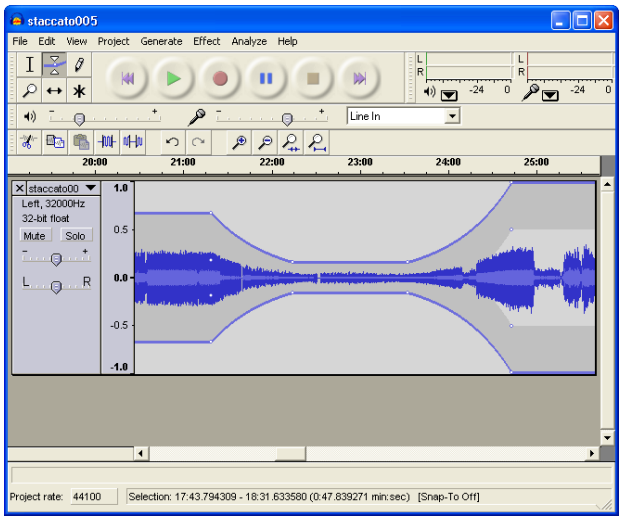


Figure 8-15 A sound sample being edited in Audacity

Sound Samples and Content

As far as an XNA game is concerned, a sound is just another item of content. It can be added alongside other content items and stored and managed by the content manager.

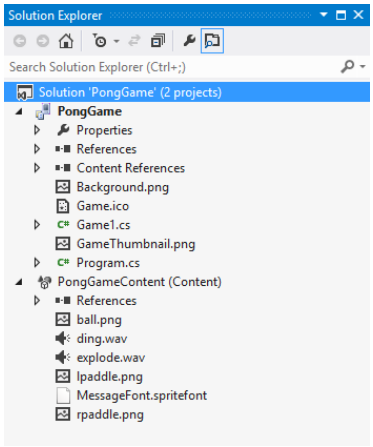


Figure 8-16 Adding sound sample content

Within the game the `SoundEffect` class is used to hold the sound items:

```
// Sound effects
SoundEffect dingSound;
SoundEffect explodeSound;
```

These are loaded in the `LoadContent` method, as usual:

```
dingSound = Content.Load<SoundEffect>("ding");
explodeSound = Content.Load<SoundEffect>("explode");
```

A `SoundEffect` instance provides a `Play` method that triggers the playback of the sound:

```
if (ballRectangle.Intersects(rPaddleRectangle))
{
    ballXSpeed = -ballXSpeed;
    dingSound.Play();
}
```

This is the code that detects a collision of the bat with the ball. The ding sound effect is played when this happens. This is the simplest form of sound effect playback. The sound is made the instant the method is called. The program will continue running as the sound is played. Windows Phone can play multiple sound effects at the same time; the hardware supports up to 64 simultaneous sound channels.

Using the SoundEffectInstance Class

The `SoundEffect` class is very easy to use. It is a great way of playing sounds quickly. A game does not have to worry about the sound once it has started playing. However, sometimes a game needs to do more than just play the sound to completion. Some sound effects have to be made to play repeatedly, change in pitch or move to the left or the right. To get this extra control over a sound a game can create a `SoundEffectInstance` value from a particular sound effect. We can think of this as a handle onto a playing sound. For example we might want to create the sound of a continuously running engine. To do this we start with a `SoundEffect` which contains the sound sample.

```
SoundEffect engineSound;
```

This would be loaded from an item of content which contains the engine sound. Next we need to declare a `SoundEffectInstance` variable:

```
SoundEffectInstance engineInstance;
```

We can ask a `SoundEffect` to give us a playable instance of the sound:

```
engineInstance = engineSound.CreateInstance();
```

Now we can all methods on this to control the sound playback of the sound effect instance:

```
engineInstance.Play();
...
engineInstance.Pause();
...
engineInstance.Stop();
```

Note that the program can stop and start the playback of the sound effect instance, which is not something that is possible with a simple sound effect. A game can also control the volume, pitch and pan of a sound effect instance:

```
engineInstance.Volume = 1; // volume ranges from 0 to 1

engineInstance.Pitch = 0;  // pitch ranges from -1 to +1
                           // -1 – octave lower
                           // +1 – octave higher

engineInstance.Pan = 0;    // pan ranges from -1 to +1
                           // -1 – hard left
                           // +1 – hard right
```

The program can also make a sound loop, i.e. play repeatedly:

```
engineInstance.IsLooped = true;
```

Finally a program can test the state of the sound playback:

```
if (engineInstance.State == SoundState.Stopped)
{
    engineInstance.Play();
}
```

This would start the sound playing again if it was stopped. Using a `SoundEffectInstance` a game can make the pitch of the engine increase in frequency as the engine speeds up and even track the position of the car on the screen.

A `SoundEffectInstance` is implemented as a handle to a particular sound channel on the Windows Phone device itself. We have to be careful that a game doesn't create a very large number of these because this might cause the phone to run out of sound channels.

The solution in *Demo 07 Game with Sounds* contains a Windows Phone XNA game that implements a working pong game with sound effects.

8.5 Managing screen dimensions and orientation

We have already seen that a Windows Phone program can be used in more than one orientation. By default (i.e. unless we say otherwise) an XNA game is created that is played in landscape mode with the display towards the left. However, sometimes we want to play games in portrait mode. Our games can tell XNA the forms of orientation they can support and then receive events when the player tips the phone into a new orientation.

The graphics class in a game provides a number of properties that games can use to manage orientation and screen size:

```
graphics.SupportedOrientations = DisplayOrientation.Portrait |
    DisplayOrientation.LandscapeLeft |
    DisplayOrientation.LandscapeRight;
```

To allow a particular orientation we just have to add it to the values that are combined using the arithmetic OR operator, as shown above.

If we want our game to get control when the orientation of the screen changes we can bind a method to the event handler as shown below:

```
Window.OrientationChanged +=
       new EventHandler<EventArgs>(Window_OrientationChanged);
```

The `Window_OrientationChanged` method can then resize all the objects on the screen to reflect the new orientation of the game. We have already seen how a program can get the width and height of the display screen. The orientation handler method would use these values to set the new dimensions of the elements in the game.

```
void Window_OrientationChanged(object sender, EventArgs e)
{
    // resize the objects here
}
```

The required orientation is best set in the constructor for the game class.

Programmer's Point: Don't worry about orientation

Very few games work in anything other than one orientation (and probably resolution). There are good reasons for this, in that designing a game that works in both portrait and landscape orientation is very hard. For that reason I'd advise you to pick an orientation that you like, and use this only. The player will not care, if the rest of the game gives a good experience.

Selecting a Screen Size

Games can also select a specific screen size:

```
graphics.PreferredBackBufferWidth = 480;
graphics.PreferredBackBufferHeight = 800;
```

If our game does this it forces the display to run at the requested resolution. It also forces the orientation too. The above width and height values would make the game work on portrait mode because the height of the game is greater than the width.

The clever thing about this is that when we set a particular display resolution the Windows Phone display hardware will make sure that we get that resolution, irrespective of the dimensions of the actual screen. In other words our game can operate as if the screen is that size and the hardware will make sure that it looks correct. This is a way you can make sure that your games will always look correct on the current version, and any new versions, of the Windows Phone hardware. No matter what new devices come along and whatever screen sizes they have the game will always look correct because the display will be scaled to fit the requested size.

We can use this to our advantage to improve the performance of our games:

```
graphics.PreferredBackBufferWidth = 240;
graphics.PreferredBackBufferHeight = 400;
```

The above statements ask for a screen which is smaller than the one fitted to most Windows Phone devices. In fact this screen is a quarter the size of the previous one. However, the hardware scaling in the Windows Phone will ensure that the screen looks correct on a larger display by performing scaling of the image. In a fast moving game this is not usually noticeable and the fact that the display is only a quarter the size of the previous one will make a huge difference to the graphical performance.

Using the Full Screen

At the moment every XNA game that we have produced has been scaled slightly to fit a smaller screen. This is because by default an XNA game does not use the top bar of the phone display. This display area is set aside for status messages. If we want our game to use the entire screen we must explicitly request this:

```
graphics.IsFullScreen = true;
```

A game can set this property to true or false. If it is set to true this means that the game can use the whole of the Windows Phone screen. I always set this to true to give my games the maximum possible amount of display area.

Disabling the Screen Timeout

The owner of a Windows Phone can set a screen timeout for the device. The idea is that if the phone is left doing nothing for a while it will shut down the screen and lock itself to save battery life. The phone monitors the touch screen to detect user input and if there is no input for the prescribed time it will shut down. The phone does not check the accelerometer however, so if we make a game that is entirely controlled by tipping the phone our customers would become very upset when they played the game. They would just be getting to an interesting part and then find that their screen went blank.

A game can disable the screen timeout function by turning off the screen saver in the XNA Guide:

```
Guide.IsScreenSaverEnabled = false;
```

The XNA Guide is the part of XNA that has conversations with the player about their Xbox Live membership and achievements etc. If we tell it to stop the screen saver from appearing this means that the game can continue until the phone battery goes flat. You should use this option with care. If a user gets bored by your game and puts the phone down it will not shut down if the screensaver has been turned off. This may mean that the phone battery will go flat as the game has been left running.

If you do have a game that is controlled by the accelerometer I would also add some gameplay element where a player has to touch the screen every now and then to keep the phone awake. I would much rather have a game that worked in this way rather than one that ran the risk of flattening the phone battery.

The best place to set these options is the constructor for the Game1 class that holds the game. This is because changes to the settings in later methods may not work correctly, as some game elements may have been configured before our settings are applied.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    graphics.SupportedOrientations =
                        DisplayOrientation.LandscapeLeft;
    graphics.IsFullScreen = true;
    graphics.PreferredBackBufferWidth = 400;
    graphics.PreferredBackBufferHeight = 240;
    Content.RootDirectory = "Content";
    Guide.IsScreenSaverEnabled = false;

    // Frame rate is 30 fps by default for Windows Phone.
    TargetElapsedTime = TimeSpan.FromTicks(333333);
}
```

This constructor sets the game up to run in a 400x240 pixel resolution in LandscapeLeft only orientation, running full screen and with no screen saver.

The solution in *Demo 08 Full Screen Pong* plays pong on a lower resolution screen that covers the entire phone surface. It is noticeable that from a player's point of view it is hard to tell from the higher resolution version.

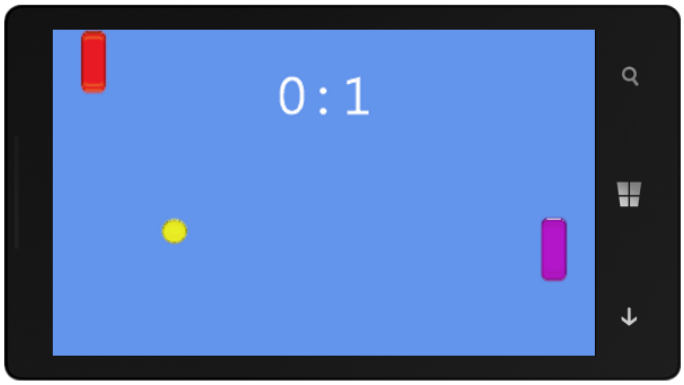


Figure 8-17 Running a low resolution game full screen

Programmer's Point: Worry about performance

The most important thing about a game is that it should be fun to play. And one of the biggest factors in fun is how fast and fluid the display is. For that reason you should worry about performance from the start. You should also make sure that you test the game on as many Windows Phone devices as you possible can, including the low cost, low performance versions.

8.6 Using MonoGame

You can still use the XNA 4.0 project type in Visual Studio 2012 to develop XNA 4.0 games and these can be sold in the Windows Phone Store to all Windows Phone users. However, Microsoft have ceased development on the XNA framework and so there is no guarantee that it will be available in future Visual Studio versions.

The good news though is that a team of Open Source developers have created a portable version of XNA which runs on Windows Phone 8, Windows 8 and also Android, iOS (Apple) and many other platforms. In this section we will look at porting an XNA Game from XNA 4.0 to MonoGame.

Note: The Windows Phone 8 and Windows 8 versions of MonoGame run as C# programs on top of the .NET infrastructure provided as part of the Windows platform. If you want to run MonoGame on non-Windows platforms you will have to obtain software that allows you to do this. The MonoGame project page has details of this.

Installing MonoGame

MonoGame can be obtained from the project web site:

```
http://www.monogame.net/
```

From there you can follow links to the latest versions of the framework and keep up to date on developments on this fast moving project.

This chapter makes use of the latest stable release at the time of writing, MonoGame 3.0.1.

To install it, visit the MonoGame site, visit the downloads page on the MonoGame site and download the latest version. Your browser (in my case Internet Explorer) will offer you the chance to run or save the installer.

Figure 8-18 Running the MonoGame Installer

You will get a warning about installing the software. Click OK and you will be presented with first dialog in the Installer.

Figure 8-19 MonoGame setup wizard

Click Next to continue the installation. You should leave all the checkboxes selected so that all the MonoGame options are installed.

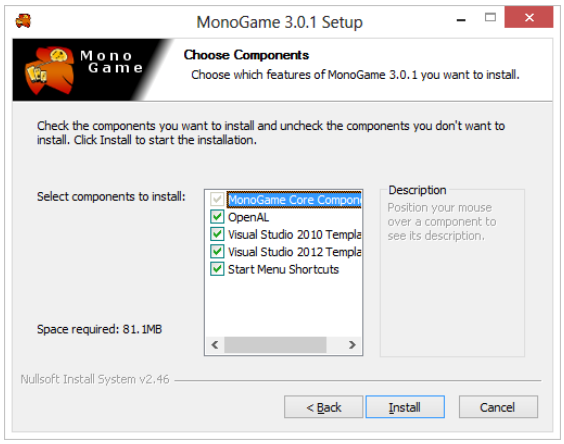


Figure 8-20 Selecting components to install

It is best just to install all the elements. **Note that you must have installed Visual Studio Express 2012 for Windows Phone before you run this installer.** When the MonoGame installer has finished you will have the MonoGame frameworks on your computer.

8.7 Making a MonoGame XNA program

After you have installed the MonoGame framework you will find some new project types available for Visual C# projects when you create a new solution.

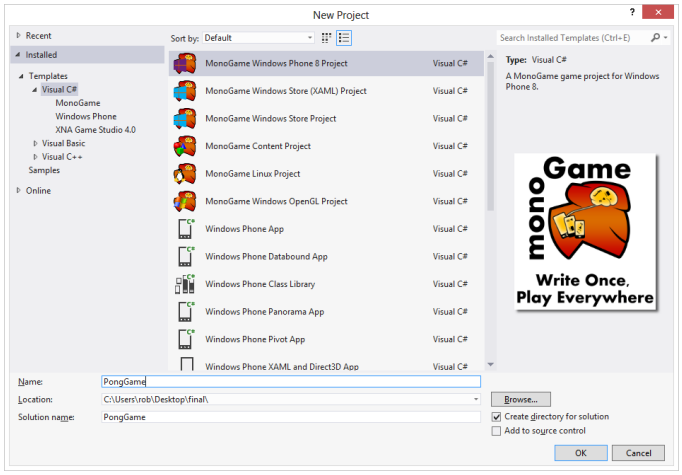


Figure 8-21 Creating a new MonoGame project for Windows Phone 8

We are going to start with a simple MonoGame Windows Phone project, so we select the top option. The Solution for an XNA project looks rather like a XAML one, but there are some differences.

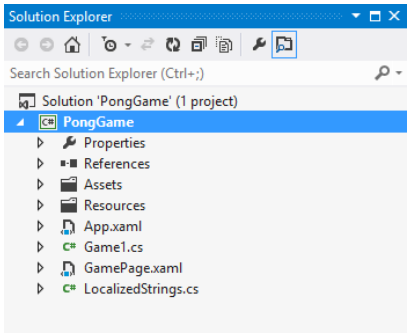


Figure 8-22 A new MonoGame solution

There are some folders for assets and resources, and the Game1.cs source file that contains the game project. However, you should note that there is no Content Manager project. The present version of MonoGame can make use of content prepared for XNA games, but it cannot create the content files. We will find out how to get content into our MonoGame projects a little later in this section.

Running a MonoGame program

Figure 8-23 An empty game

If you run the new game project as created above you will be rewarded by a blue screen as shown in Figure 8-23. The numbers down the side of the game screen are performance counters that give you the frame rate and other useful items.

Note: Because of the way that the Windows Phone emulator is implemented for Windows Phone 8 it is not possible to run games in the emulator on the Windows PC. Instead you must run the games you create on a device that you have attached

to your PC. This is actually a good thing in a way, as it makes it much easier for you to test your games and get a feel for what the game is really like to play.

Content in MonoGame solutions

We have seen that in XNA 4.0 you can just add images and sound files to the content project in the game solution and these are automatically converted to XNA resources that can be used in games. The idea behind content conversion is that the game running in the target device doesn't have to have decoders for all the different types of image and sound files that are available, it just has to be able to read the XNA format files, which have the language extension "xnb". If you take a look at the output directory from one of the Pong games that we created earlier you can see all these files that have been created when the solution was built.

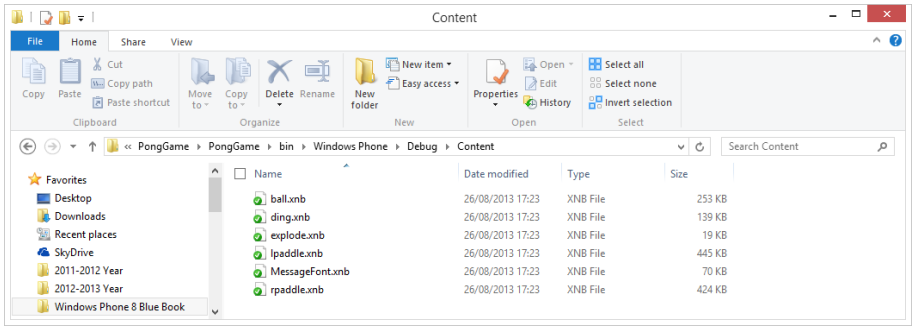


Figure 8-24 The xnb content files in a game distribution

The Content Loader in a MonoGame application can read these files and create textures and sound effects from them but at the moment there is a Content Loader for MonoGame games.

This means that there is no way of creating the xnb files from within a MonoGame solution. Instead you have to get the actual xnb files from an XNA 4.0 project and add them by hand to a Content folder in the MonoGame solution. This is not actually very difficult. You just have to navigate to the xnb file (you can see where it is on the path above) and then drag files from the folder into a Content folder that you create in the MonoGame project.

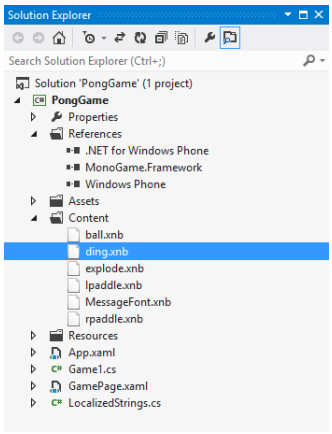


Figure 8-25 The xnb files installed as content

These resources can then be loaded exactly as they would be in an XNA 4.0 game.

```
font = Content.Load<SpriteFont>("MessageFont");
```

Note that you don't have to add the .xnb extension, and that this is also how you import font files as well as other types of resource.

Each resource file should be set as Content and it should be copied into the output folder when the project is built.

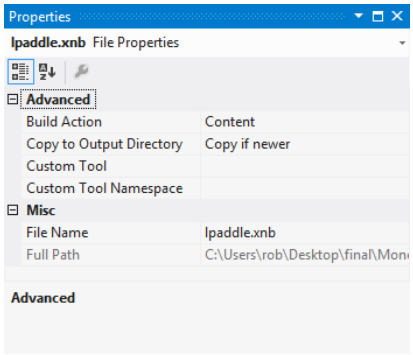


Figure 8-26 The xnb file properties

If you are creating a MonoGame solution alongside an XNA 4.0 one you may want to add the content as links rather than copies of the original item. You can do this by adding the resources as a link rather than just referring to the resource file. This means that any changes to the resource will be reflected next time you build the MonoGame one.

Figure 8-27Adding a Link to a Visual Studio Project

You select an input as a link by using the drop down setting on the import command, as show in Figure 8-27 above.

The solution in *Demo 09 MonoGame Pong* is an implementation of the Pong game in MonoGame.

Working with MonoGame

The MonoGame framework on Windows Phone 8 should be regarded as a bit of a "work in progress" at the moment. For example, there are issues with the sound output in the current release of the framework, although these can be resolved by updating some of the components. However, it is very easy to migrate an XNA 4.0 solution into MonoGame. My advice at the moment would be to develop for XNA 4.0 to start with, and then migrate your solution at the end. This also gives you the benefit that you can use the Windows Phone emulator to test your games on your PC.

Having said this, MonoGame offers a fantastic opportunity for you to take your games and move them onto wide range of different platforms.

What We Have Learned

1. The XNA framework provides an environment for creating 2D and 3D games. The games can be created for Windows PC, Xbox 360 or Windows Phone 7.5.
2. The MonoGame framework allows you to take XNA games and run them on iOS, Android, Windows 8 and Windows Phone 8 platforms.
3. XNA games operate in a completely different way from XAML applications. An XNA game contains methods to load content, update the game world and draw the game world which are called by the XNA framework when a game runs.
4. When Visual Studio creates a new XNA game it makes a class which contains empty LoadContent, Update and Draw methods that are filled in as a game is written. The LoadContent method should load all the required game content. The Update method is called by XNA 30 times a second to update the game content and the Draw method is called to render the game on the screen.
5. Any content (for example images or sounds) added to an XNA game is managed by the Content Management process that provides input filters for different file types and also output processors that prepare the content for deployment to the target device. Within an XNA game the way that content is loaded and used the same irrespective of the target hardware.
6. A sprite is made up of a texture which gives the image to be drawn and a position which determines where to draw the item and its size. The XNA Framework provides objects that can hold this information.

7. When drawing in 2D within the XNA system a given position on the screen is represented by pixel coordinates with the origin at the top left hand corner of the display.
8. The Windows Phone touch panel can provide a low level array of touch location information which can contain details of at least 4 touch events.
9. XNA games can display text by using spritefonts made from fonts on the host Windows PC
10. XNA programs can play sound effects. For greater control of sound effect playback a program can create a SoundEffectInstance value which is a handle to a playing sound.
11. XNA programs on Windows Phone can select a particular display resolution and orientation. If they select a resolution lower than that supported by the phone display the GPU will automatically scale the selected size to fit the screen of the device.
12. XNA programs on Windows Phone can disable the status bar at the top of the screen so that they can use the entire screen area. They can also disable the screensaver so that they are not timed out by the phone.

9 Using Speech in Applications

The Windows Phone operating system provides very good support for speech. The phone can be made to speak text and also recognise what the user has said. In this section we are going to take a look at how we can make programs that speak, start programs under voice control and finally respond to spoken commands.

9.1 Speech Synthesis

Windows Phone 8 provides extremely high quality voice output. It is very easy to make your programs speak to the user. The speech output can be via the speaker built into the phone or via a wired or Bluetooth headset.

Figure 9-1 Enabling the speech recognition capability.

Note that you have to enable this capability for speech output, as well as input. If you want to make your programs easier to write you can also bring in the speech synthesis namespace.

```
using Windows.Phone.Speech.Synthesis;
```

To output speech a program just has to create a speech synthesizer and then use it:

```
SpeechSynthesizer synth = new SpeechSynthesizer();

await synth.SpeakTextAsync("Hello World");
```

The first statement creates a speech synthesizer using the default language. The second statement uses this to say the phrase "Hello world".

Note that the call of `SpeakTextAsync` is preceded by the `await` keyword. This is because the method executes *asynchronously*. If you've not seen this before, it might seem confusing.

The `await` keyword is used with methods that take some time to complete. We first saw it when we were looking at network transactions in chapter 7. A network request can take a while to finish, and so can a speech request. We are going to write a little application that speaks output to the user:

Figure 9-1 Simple Speaker

The user types in text and presses the Speak button. The phone than speaks what was entered. You could use it to order coffee if you had a bad case of laryngitis. The button event handler creates a speech synthesizer and then speaks the text:

```
async private void speakButton_Click(object sender,
                                     RoutedEventArgs e)
{
    SpeechSynthesizer synth = new SpeechSynthesizer();

    statusTextBlock.Text = "Speaking";

    await synth.SpeakTextAsync(speechTextBox.Text);

    statusTextBlock.Text = "Finished";
}
```

The method runs when the button is pressed to start the speech. Normally we would have to create a thread to do the actual speaking, as one of the rules of user interface implementation is that code in event handlers should complete very quickly. If the button event handler was held up while the speech was being produced this would result in a very unresponsive user interface.

However, the above method works fine without threads. The Speak button becomes useable before the speech has completed, so that a user can "queue up" a number of speech requests by repeatedly pressing the button.

This is because of the use of `await` and `async`. The `await` keyword is used to flag methods that must be called asynchronously. When the compiler sees such a method it generates code that will create a task to run that method and the code that follows it. It then returns instantly to the caller of the `async` method. This means that the speak button will be enabled before the speak method has completed.

If the user presses the button before the system has finished speaking a message the system will simply create another task that will run after the first one has finished.

You can actually see this in action. When you run the program and press the speak button the status changes to "Finished" after the program has finished speaking.

The solution in *Demo 01 SimpleSpeaker* contains an application will speak the things that you enter. It also numbers each speech request and displays these numbers when speech requests complete, so you can see how successive calls to the speech method are queued up and resolved if you rapidly press the Speak button.

Programmer's Point: Await and Async do not solve all your problems

Please remember that just because these new keywords make it easier to create and use threads and tasks that they don't solve all your problems. In fact, if anything, they make things even more dangerous. For example, consider what would happen if an asynchronous speak method used a class variable to tell it what to say. If that variable was not managed in a thread safe way (for example if other tasks were able to change it) then this may result in the program not saying what was expected, as another task had changed the variable from the value it had when the speak method was initiated. Make sure that if you create software that is designed to work in a "thread safe" way if you use these keywords in your programs.

Speech Exceptions

It turns out that there are occasions when a speech request can throw an exception. This has to do with the way that a user can switch from one program to another at any time. If a user navigates away from your program, perhaps by pressing the Start or Back buttons any speech the program is generating is abandoned. You might think that this is the point at which an exception would be produced to tell your program that this has happened, but of course this is also the point at which your program is being shut down (we will talk about what happens in these situations later in the text) and so there is no point in telling your program that this has happened.

Instead, what happens is that the exception is generated if your program is ever resumed, i.e. if/when the user returns to it. The exception type that identifies this event is a specific one that your program can test for.

```
bool beenInterrupted = false;

async private void speaker(string message)
{
    beenInterrupted = false;
    try
    {
        SpeechSynthesizer synth = new SpeechSynthesizer();
        await synth.SpeakTextAsync(message);
    }
    catch ( Exception ex)
    {
        if (((uint)ex.HResult == 0x80045508))
        {
            // System Call Interrupted thrown by Speech
            beenInterrupted = true;
        }
    }
}
```

Above you can see a speaker method that is called to deliver a spoken message. The method catches the speech exception and sets a flag, `beenInterrupted`, to indicate that the user moved away from the application during the speech.

It is very important that any program that produces speech catches this exception as a program that doesn't catch this exception will fail certification if it is submitted for sale in the Windows Phone Store.

The solution in *Demo 02 Speaker Method* uses a speaker method that speaks and catches any exception that may be thrown. It sets a flag to indicate if the speech was interrupted.

Selecting Different Languages

The `SetVoice` method sets the voice to be used when speaking. The `InstalledVoices` collection holds all the voices that are available on a phone. The statements below search for a French voice and then set the voice to use it.

```
// Query for a voice that speaks French.
var frenchVoices = from voice in InstalledVoices.All
                   where voice.Language == "fr-FR"
                   select voice;
// Set the voice as identified by the query.
synth.SetVoice(frenchVoices.ElementAt(0));
```

This code is rather interesting, in that it builds a query to find the required voice. Note that if the French voice is not available the above code will fail. Your program should really make sure that a particular language is available before using it.

```
foreach (VoiceInformation v in InstalledVoices.All)
{
    synth.SetVoice(v);
    await synth.SpeakTextAsync("I like cheese.");
}
```

The code above works through all the languages that are installed on a device and says "I like cheese" using each language.

Using Speech Markup Language

At the moment we have just supplied plain English text to the speech synthesizer. It will make a "best guess" attempt to produce the sound output, but we can give it some help by adding markup information to the text.

```
string str = "<speak version=\"1.0\"";
str += " xmlns=\"http://www.w3.org/2001/10/synthesis\"";
str += " xml:lang=\"en-US\">";
str += "<p> Your <say-as interpret-as=\"ordinal\">1st</say-as> request was made on ";
str += "<say-as type=\"date:mdy\"> 1/29/2013 </say-as></p>";
str += "</speak>";
synth.SpeakSsmlAsync(str);
```

The above code builds a string that contains Speech Synthesis Markup Language (SSML) and then uses a different call to speak using it. If you want to find out more about the elements you can add to a markup string you can find a detailed description here:

```
http://www.w3.org/TR/speech-synthesis/
```

There are also methods you can use to make a program speak the contents of a file.

9.2 Controlling Applications using Speech

Generating speech is very easy. You can also use the voice input on Windows Phone to control an application. The first kind of voice control we are going to look at is starting the application itself. The Windows Phone operating system has voice response built in and you can use it to start a program from the Listening prompt.

Figure 9-2 Listening for commands

Figure 9-2 shows the display produced by holding down the Windows key on the phone. You can speak any one of a number of commands, including “Open” followed by the name of a program to run. In fact, you can also leave out the word “Open” and the system will usually find the required program.

This listening interface is also available via Bluetooth and headset connections so that a phone user can control the phone without having to actually press anything.

This means that to get voice activation of your program you don’t actually have to do anything, as this is enabled automatically. However, it is also possible for you to design a set of voice commands that are passed into your program when it starts running. This makes it possible to allow interactions such as “Add Reminder fifteen minutes”. The application “Add Reminder” would be started and the time information passed directly into it. You do this by creating a Voice Command Definition or VCD file.

The Voice Command Definition File

The Voice Command Definition (VCD) file is part of the Visual Studio solution for an application. It is deployed with the application and is used by the operating system to configure the commands that will be recognised and passed into the application when that application is started. You can create a template VCD file by using the “Add New Item” dialog in Visual Studio.

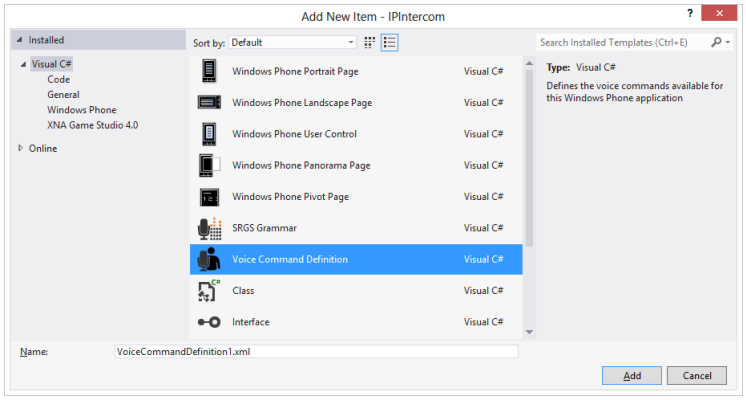


Figure 9-3 Adding a Voice Command file

Once you have added a file it should be marked as content and Visual Studio should be configured to "Copy if Newer":

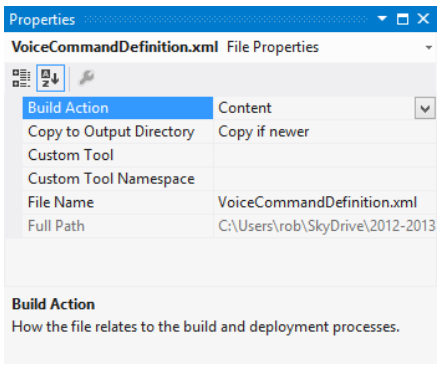


Figure 9-4 Configuring a Voice Command file

This file contains an XML structured description of the commands that can be passed into the program when it starts running.

If you want the phone to recognise and use the commands you have to explicitly load the command file when your program runs.

```
await VoiceCommandService.InstallCommandSetsFromFileAsync(
    new Uri("ms-appx:///VoiceCommandDefinition.xml", UriKind.RelativeOrAbsolute));
```

The above statement loads the voice command file. Note that it is an asynchronous method (like the ones that produce speech) as this loading can take some time. Note also that this means that the user has to run your program before the speech behaviours will work.

The VCD file identifies commands and options and uses them to determine which page of your application is loaded when the application is started. It can also set the values of information that is passed into the page.

Quick Reminder VCD commands

As an example, consider a "Quick Reminder" program. This can be used to set a reminder from speech input. You can say things like "Quick Reminder set five" and it will set a reminder that will ring in five minutes time. We will do this by creating a VCD file that has a single command, set, in it:

```
<CommandSet xml:lang="en-us">
  <CommandPrefix>Quick Reminder</CommandPrefix>
  <Example> ten minutes </Example>

  <Command Name="SetReminder">
    <Example> set ten </Example>
    <ListenFor> set {number} [minutes]</ListenFor>
    <Feedback> Setting a reminder... </Feedback>
    <Navigate Target="MainPage.xaml"/>
  </Command>

  <PhraseList Label="number">
    <Item> five </Item>
    <Item> ten </Item>
    <Item> twenty </Item>
  </PhraseList>
</CommandSet>
```

If you look carefully at this you can see how it works. The VCD file contains one command and one phraselist. The `CommandPrefix` gives what the user should say to initiate the voice commands for the program. Note that this doesn't have to be the actual name of your program, but it might be sensible to make it so.

Each command has a different name, and something that is listened for to trigger the command. When it is triggered the receiving program can identify the command by this name. A command always has example text, this is displayed by the phone when the user requests help on the command. The command also has a `ListenFor` element and a `Feedback` element. The `ListenFor` element gives the text that triggers this particular command. In the case above the trigger word is set, followed by one of the words from the number phraselist, and then an optional word, minutes.

If you look carefully you can see how this works. The system will listen for plain text in the `ListenFor` element. Text enclosed in the curly brackets {} is the name of a phraselist (in this case one of a number of possible times) and text enclosed in square brackets is optional.

The final part of the command set is the target page to navigate to when the command is triggered. I have set this to the MainPage of the application, but you can use this feature to direct the user to any one of a number of pages, depending on the command that has been given.

Using the Voice Command in the program

When the commands have been loaded the program can now be started via voice commands. When the program navigates to a particular page that page can check to see if a voice command has been used to activate it:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    setupVoiceCommands();

    // Is this a new activation or a resurrection from tombstone?
    if (e.NavigationMode == System.Windows.Navigation.NavigationMode.New)
    {
        // Was the application launched using a Voice Command?
        if (NavigationContext.QueryString.ContainsKey("voiceCommandName"))
        {
            // If so, get the name of the Voice Command.
            string voiceCommandName =
                                NavigationContext.QueryString["voiceCommandName"];

            switch (voiceCommandName)
            {
                case "SetReminder":
                    string delayNumberString =
NavigationContext.QueryString["number"];
                    int delay = lookUpNumber(delayNumberString);
                    ReminderManager.SetReminder(delay, "Quick Reminder");
                    break;
            }
        }
    }
}
```

We know that when a page in a Windows Phone application is displayed the method `onNavigatedTo` is called on that page. The first line of this method sets up the voice commands for the application. Then a test is made to see if this is a new activation of the page. What we are looking for is the situation where the application has been started from cold, i.e. the user has issued a spoken command to start it. We don't want to be resetting the reminder each time a user goes back to it.

If the page is being visited for the first time the program checks for a `voiceCommandName` key in the navigation context. This is how the Windows Phone operating system passes in the command that has been recognised. If the key exists then it is extracted and used to control a switch construction. At the moment this only has the one case, but if you had lots of commands you could add them here. The `SetReminder` command has a delay number associated with it. This is pulled out of the navigation context and used to determine the length of time before the reminder fires.

The code uses a helper method that I wrote to convert a string into a number, and then users a helper class to create the required reminder. If you want to find out more about how to create a reminder, take a look in Chapter 14.

The solution in *Demo 03 Quick Reminder* contains a working Quick Reminder program that you can use to set reminders using voice. It is a bit rough and ready, in that it does not have a lot of error handling, but it does illustrate the principles and provides a good basis for your own voice controlled applications.

Languages and Voice Control Files

The VCD file actually contains multiple language versions of the control files. Each of these has a language code that is checked to make sure that the VCD commands match the language in use. If there is no command file available for the language setting of a phone the request to load the voice command file will throw an exception.

A program can check the current language in use as follows:

```
string language = CultureInfo.CurrentCulture.Name;
```

The `CultureInfo` class lives in the `System.Globalisation` namespace and can be used to get the name of the language in use. There must be a set of Voice Commands which match this name in the VCD file:

```
<?xml version="1.0" encoding="utf-8"?>

<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.0">
  <CommandSet xml:lang="en-us">
    <CommandPrefix>Quick Reminder</CommandPrefix>
    <Example> ten minutes </Example>

    <Command Name="SetReminder">
      <Example> set ten </Example>
      <ListenFor> set {number} [minutes]</ListenFor>
      <Feedback> Setting a reminder... </Feedback>
      <Navigate Target="MainPage.xaml"/>
    </Command>

    <PhraseList Label="number">
      <Item> five </Item>
      <Item> ten </Item>
      <Item> twenty </Item>
    </PhraseList>
  </CommandSet>

  <CommandSet xml:lang="en-gb">
                                ...
                                GB English version
                                ...
                            </CommandSet>
                        </VoiceCommands>
```

Above you can see a complete VCD file. It is made up of VoiceCommands sections, one for each language, with a lang property for each section. If there is not a command section to match the selected language on the phone the VCD file will not load successfully.

A program can check to see if a particular language is installed in the currently active VCD file.

```
if (!VoiceCommandService.InstalledCommandSets.ContainsKey(language))
{
    MessageBox.Show("Voice Commands not available");
}
```

This code snippet will display a message box if the required voice commands are not available. Note that this will be the case the very first time a program runs.

Modifying Voice Control Files

An application can load and reload the VCD file at any time. It is also possible to programmatically update items in an existing VCD file

```
string[] number = new string[] { "one", "two", "three" };

VoiceCommandSet activeSet =
          VoiceCommandService.InstalledCommandSets["en-gb"];

await activeSet.UpdatePhraseListAsync("number", number);
```

The code above would change the number phraselist to contain the strings “one”, “two” and “three”. Note that the previous phrase list would be replaced and there is no way of reading back the phrases in a list that has been loaded. There is an upper limit of 2,000 on the number of items you can have in a phraselist.

9.3 Simple speech input

You can also make your applications recognise speech. The speech is returned as a string you can then just use in your program.

To allow speech recognition you have to, obviously, enable the IO_CAP_SPEECH_RECOGNITON capability, but you also have to enable IO_CAP_MICROPHONE.

```
SpeechRecognizerUI recoWithUI;

async private void listenButton_Click(object sender, RoutedEventArgs e)
{
    recoWithUI = new SpeechRecognizerUI();

    SpeechRecognitionUIResult recoResult =
        await recoWithUI.RecognizeWithUIAsync();
    if (recoResult.ResultStatus == SpeechRecognitionUIStatus.Succeeded)
        MessageBox.Show("You said " + recoResult.RecognitionResult.Text);
}
```

The above code snippet will recognise some speech and then display it in a message box.

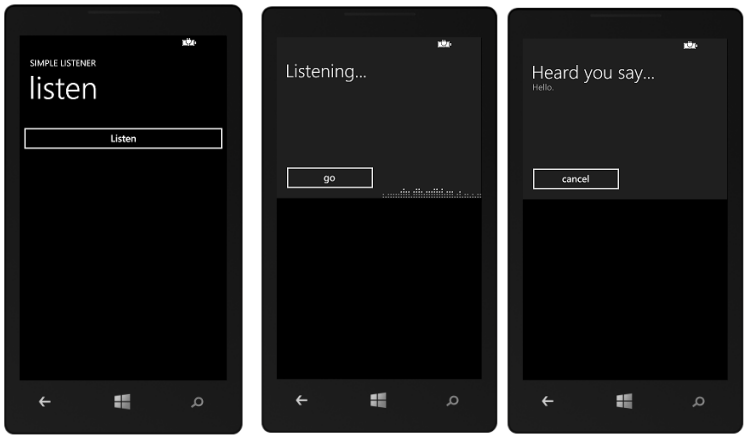


Figure 9-5 Voice recognition UI

Figure 9-5 shows a simple voice recognition program in action. When the user presses the Listen button the code above runs, recognises the words and then displays them.

The solution in *Demo 04 Simple Listener* contains a program that will recognise speech and then display it.

Customising Speech Recognition

A recogniser can be configured with additional options, for example:

```
recoWithUI.Settings.ReadoutEnabled = false; // don't read the saying back
recoWithUI.Settings.ShowConfirmation = false; // don't show the confirmation

recoWithUI.Recognizer.Settings.InitialSilenceTimeout = TimeSpan.FromSeconds(6.0);
recoWithUI.Recognizer.Settings.BabbleTimeout = TimeSpan.FromSeconds(4.0);
recoWithUI.Recognizer.Settings.EndSilenceTimeout = TimeSpan.FromSeconds(1.2);
```

The first two options are self-explanatory. The others are used as follows:

- **InitialSilenceTimeout:** the time that the speech recognizer will wait until it hears speech. The default setting is 5 seconds.
- **BabbleTimeout:** the time that the speech recognizer will listen while it hears background noise. The default setting is 0 seconds (the feature is not activated).
- **EndSilenceTimeout:** the time interval during which the speech recognizer will wait before finalizing the recognition operation. The default setting is 150 milliseconds.

It is worth spending some time tuning these values, depending on how your application is going to use voice response.

Recognising without a user interface display

An application can recognise speech without displaying a user interface:

```
SpeechRecognizer recoNoUI;

async private void listenButton_Click(object sender, RoutedEventArgs e)
{
    recoNoUI = new SpeechRecognizer();

    SpeechRecognitionResult recoResult =
        await recoNoUI.RecognizeAsync();
    if (recoResult.TextConfidence == SpeechRecognitionConfidence.High ||
        recoResult.TextConfidence == SpeechRecognitionConfidence.Medium)
        MessageBox.Show(string.Format("You said " + recoResult.Text));
    else
        MessageBox.Show("Pardon?");
}
```

This version of the listener doesn't display any user interface. The program can use a confidence value to determine how successful the recognition was.

The solution in *Demo 05 Simple Listener No UI* contains a program that will recognise speech and then display it. This version does not display a user interface.

It is also possible to bind to events generated by a SpeechRecognizer, there are events for both successful and unsuccessful recognition.

Configuring Speech Recognition

The speech recognition settings are configured from the phone settings menu.

Figure 9-6 Speech Recognition settings

Users can select the language to use and also turn the speech recognition service on or off. As we shall see in the next section, using free speech recognition will result in extra network traffic, and the user has the option to turn this feature off.

Speech Recognition and Network Access

The Voice Command Definition (VCD) file recognition process does not require a network connection to work. Because the voice recognition has a fixed set of words to recognise it is possible for the phone to do a very good job of determining which commands have been entered. This means that users can start applications and select options without the need for a network connection.

Unfortunately, the same is not true when recognising free speech. In this situation the phone software is not able to perform the recognition and so it sends the speech data to a server which then returns with the converted text. There is no charge for this service and it works transparently as far as the phone user is concerned.

However, for this to work the phone must have a working network connection at the time the conversion is to be performed. If this is not available the speech recognition request will fail. If your application makes use of free speech recognition you must bear in mind that it will only be useable when a network connection is available.

9.4 Using grammars

Grammars provide a way in which an application can have a structured conversation with the user. They have the advantage that they do not require a network connection to recognise the spoken input. Because the grammar sets out the entire vocabulary of a conversation the phone can perform recognition directly, with no need for a network connection.

Grammars can be created using the Speech Recognition Grammar Specification (SRGS) Version 1.0 and stored as XML files loaded when the application runs. This is a little complex, but worth the effort if you want to create applications with rich language interaction with the user – there are examples on MSDN that you can take a look at

```
http://msdn.microsoft.com/en-us/library/hh361658
```

We are just going to investigate a simple grammar that allows a user to pick from a range of options, in this case their preferred strength of cheese. The grammar will contain a list of possible replies along with the question that is to be asked.

```
recoWithUI = new SpeechRecognizerUI();

// Build a string array, create a grammar from it, and add it to
// the speech recognizer's grammar set.

string[] strengthNames = { "weak", "mild", "medium", "strong",
                           "english" };
    recoWithUI.Recognizer.Grammars.AddGrammarFromList("cheeseStrength",
                                                      strengthNames);

recoWithUI.Recognizer.Grammars["cheeseStrength"].Enabled = true;
```

These statements create a new recogniser and build a list of options from which a selection is to be made. The list is added as a grammar to the recognizer and then enabled. Note that the recognizer uses a dictionary based structure to allow you to add lots of grammars, select them and enable or disable them.

Once the grammar has been set up the program can then ask a question.

```
recoWithUI.Settings.ListenText =
                           "How strong do you like your cheese?";

try
{
    SpeechRecognitionUIResult recoResult =
                         await recoWithUI.RecognizeWithUIAsync();

    if (recoResult.RecognitionResult.TextConfidence ==
                              SpeechRecognitionConfidence.High)
    {
        MessageBox.Show("Cheese: " +
                            recoResult.RecognitionResult.Text);
    }
    else
    {
        MessageBox.Show("Sorry, didn't get that");
    }
}
catch
{
}
```

The above statements make a request and then display the result of the question if the speech recognition has high confidence in the response.

Figure 9-7 Asking about cheese

The solution in *Demo 06 AskAboutCheese* contains a program that will ask you about your cheese preference and display the result.

What We Have Learned

1. A Windows Phone application can be started by speech and can produce speech output and respond to speech input.
2. There are a large number of voice and language options for speech output.
3. Speech output is asynchronous, the speech methods can be called in a way that allows them to run in the background without any need for the developer to create and manage tasks and threads.
4. By using the Speech Synthesis Markup Language (SSML) a developer can get a greater level of control over the speech output process.
5. An application can be associated with a Voice Command Definition (VCD) file that identifies commands and options that can used to customise the behaviour of the application when it is started. This customisation can be used to pass data into the program and select the XAML page to be loaded when the application starts running.
6. Applications can recognise free speech, either with a user interface or in the "background". Programs are given a string that has been created from the speech input, along with information giving the confidence level of the recognition system. **This service makes use of the network connection on the phone as the speech data is not processed on the device. This means that free speech recognition is not available if the phone is not connected to a network.**
7. Voice recognition can also be performed by creating pre-built grammars which allow a program to recognise spoken commands in a particular format. Grammar based speech recognition does not require the use of a network connection to perform the speech recognition.

10 Maps and Location

All Windows Phone devices contain a Global Positioning System (GPS) receiver and are also able to use environmental data (for example locations of mobile phone cell towers and Wi-Fi connections) to determine their position to a high level of accuracy. This location information can be used in conjunction with the mapping services provided by Bing Maps to allow you to create applications which use location.

In this chapter we are going to take a look at how a program can determine the position of a phone and then work with the mapping services to create location aware applications.

10.1 Determining the geoposition of the phone

Any application that makes use of the location of the phone must enable this capability. The user will be made aware of the fact that the application will be using location data and asked to confirm the installation of your application. This is to prevent malicious applications using location data without the agreement of the user.

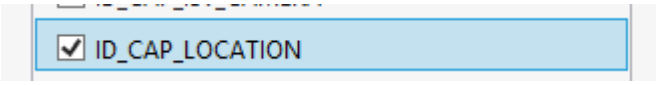


Figure 10-1 Enabling the location capability

The `Geolocator` class can be used to determine the location of a phone. It is found in the `Windows.Devices.Geolocation` namespace.

```
Geolocator locator = null;
```

A program can create an instance of a `GeoLocator` and then use it whenever the location of the phone is required.

```
if (locator == null)
{
    locator = new Geolocator();
}
```

This creates a new `locator` instance if one does not already exist. We can then use this to determine the position of the phone:

```
async private void findMeButton_Click(object sender,
                                      RoutedEventArgs e)
{
    Geoposition position = await locator.GetGeopositionAsync();

    latTextBlock.Text = "Latitude: " +
                        position.Coordinate.Latitude.ToString();
    longTextBlock.Text = "Longitude: " +
                         position.Coordinate.Longitude.ToString();
}
```

The method above runs when the user presses an XAML button which requests the location of the phone. Note that this is an asynchronous method, which means that although the button event handler will return immediately the `GetPositionAsync` method may take some time to complete. This is reasonable as it may be necessary for the system to start the GPS system running and then wait for it to acquire position information. This can take several seconds to complete. The GPS system on a Windows Phone is one of the most power hungry interfaces on the device, and so it is not active all the time.

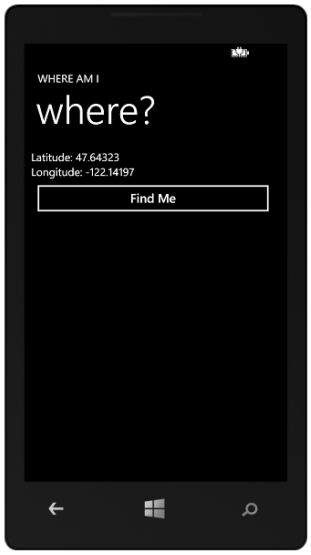


Figure 10-2 Simple location display

The location is delivered in the form of double precision values that give the latitude and longitude of the phone position.

Phone position in the Emulator

The position in Figure 10-2 shows the default position of the Windows Phone Emulator

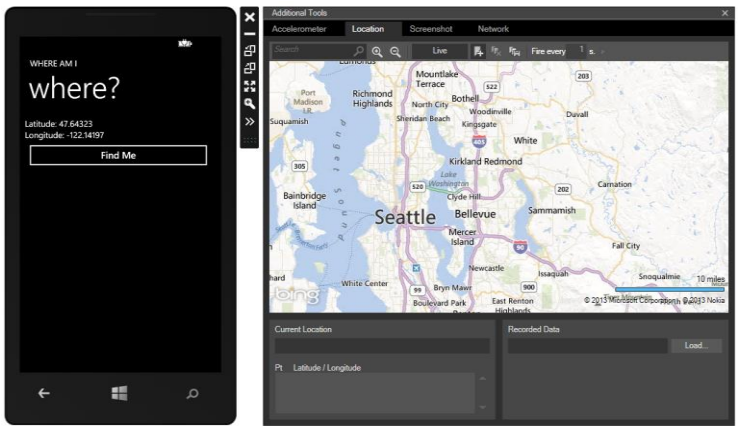


Figure 10-3 Setting position in the emulator

If you click the right pointing arrow at the bottom of the Emulator control panel you can open up the Additional Tools menu for the emulator which lets you set the "location" of the emulator by clicking on the map. By default the phone is positioned in Seattle. You can create paths on the map which the emulator will then traverse. This makes it very easy to design routes that the emulator will follow while you test your programs.

The appropriate position information is delivered each time your application requests it. You should note however that this position emulation is not available to background applications (of which more later) which want to use the position information.

Setting the required precision

A program can select the required precision of the location information by adjusting a property on the `Geolocator` instance:

```
if (locator == null)
{
    locator = new Geolocator();
    locator.DesiredAccuracy = PositionAccuracy.High;
}
```

The initial setting for the accuracy is `Default`. Note that if you use the high precision setting this will have implications for battery life. Unless you have a really good reason for having the best possible location information you should leave the setting at `Default`.

Determining the source of the position information

The position data also contains a property indicating the source of the position information.

```
switch (position.Coordinate.PositionSource)
{
    case PositionSource.Cellular:
        break;
    case PositionSource.Satellite:
        break;
    case PositionSource.WiFi:
        break;
}
```

Your program can then behave appropriately. Note that Cellular position data will vary in quality depending on the number of transmitter towers that are visible to the phone and that the Wi-Fi accuracy depends on the position information for the local networks. Note also that the emulator will generate a position source of `Cellular` if the `PositionAccuracy` is set to `Default` and `Satellite` if it is set to `High`.

The solution in *Demo 01 Where Am I* contains a program that will display the location of the phone when the button is pressed.

The above code that reads the phone position is not very good in that if location information is not available (for example the phone is in Airplane mode and unable to receive from satellites) the call to `GetPositionAsync` will take a long time to complete. In the case of the Where Am I program this means that the user will repeatedly press the button. This will not cause the program to fail, it just means that a large number of reads requests will be queued up. We can address this problem by turning off the button once it has been pressed:

```
async private void findMeButton_Click(object sender,
                                      RoutedEventArgs e)
{
    findMeButton.IsEnabled = false;
    Geoposition position = await locator.GetGeopositionAsync();
    sourceTextBlock.Text =
              position.Coordinate.PositionSource.ToString();
    latTextBlock.Text = "Latitude: " +
              position.Coordinate.Latitude.ToString();
    longTextBlock.Text = "Longitude: " +
              position.Coordinate.Longitude.ToString();
    findMeButton.IsEnabled = false;
}
```

This version of the button event handler disables the button before it attempts to get a position. However, it will still annoy the user if it causes the user interface to pause.

Time Outs

You can get more control over the responsiveness of request by using an alternate version of the `GetGeopositonAsync` method. This allows you to set a timeout:

```
TimeSpan acceptableAge = new TimeSpan(0, 0, 30);
TimeSpan timeOut = new TimeSpan(0,1,0);
Geoposition position =
     await locator.GetGeopositionAsync(acceptableAge, timeOut);
```

The first parameter is the “acceptable age” of the reading. The GPS system is constantly active and shares positioning information with all the processes running on the phone. It may be that there is position information available that was captured a few second ago. The program can specify how old recent position data can be. The code above will accept data which is “younger” than 30 seconds old. The second parameter is the length of time that the method call will wait for a reading before abandoning the request. Both these parameters are expressed as `TimeSpan` values.

A program can find out the time of a reading by using the `TimeStamp` property of the coordinates that are supplied:

```
timeTextBlock.Text = position.Coordinate.Timestamp.ToString();
```

There are other properties of the Coordinate that will give the resolution to which the position was determined and other useful details.

It is possible that the reading the position may throw exceptions.

```
async private void findMeButton_Click(object sender,
                                      RoutedEventArgs e)
{
    findMeButton.IsEnabled = false;
    TimeSpan acceptableAge = new TimeSpan(0, 0, 30);
    TimeSpan timeOut = new TimeSpan(0, 1, 0);
    try {
        Geoposition position =
         await locator.GetGeopositionAsync(acceptableAge, timeOut);

        timeTextBlock.Text =
                    position.Coordinate.Timestamp.ToString();
        sourceTextBlock.Text =
                    position.Coordinate.PositionSource.ToString();
        latTextBlock.Text = "Latitude: " +
                            position.Coordinate.Latitude.ToString();
        longTextBlock.Text = "Longitude: " +
                            position.Coordinate.Longitude.ToString();
    }
    catch (System.UnauthorizedAccessException) {
        timeTextBlock.Text = "not allowed";
        sourceTextBlock.Text = "";
        latTextBlock.Text = "";
        longTextBlock.Text = "";
    }    catch (TaskCanceledException) {
        timeTextBlock.Text = "cancelled";
        sourceTextBlock.Text = "";
        latTextBlock.Text = "";
        longTextBlock.Text = "";
    }
    finally {
        findMeButton.IsEnabled = true;
    }
}
```

The above code creates a “find me” button behaviour that will deal with exceptions and work well.

The solution in *Demo 02 Well behaved Where Am I* contains a program that will display the location of the phone when the button is pressed. It disables the button and also contains exception handling.

Position Events

The above code works well if you want a program to just determine the position upon request, but you often want to be able to track the location of a device. The `Geoposition` class will do this for you. An event can be fired in your program when position of the phone changes by more than a set amount:

```
locator = new Geolocator();
locator.DesiredAccuracy = PositionAccuracy.High;
locator.MovementThreshold = 20; // distance in meters
locator.PositionChanged += locator_PositionChanged;
locator.StatusChanged += locator_StatusChanged;
```

The program must set the movement threshold before binding to the position changed event. In the above code the threshold is set at 20 meters.

The locater is generating two events, one which is produced when the locator changes state, and the other is produced when the when the state of the GPS system changes.

```
void locator_PositionChanged(Geolocator sender, PositionChangedEventArgs args)
{
    Dispatcher.BeginInvoke(() =>
    {
        updateDisplay(args.Position);
    });
}

private void updateDisplay(Geoposition position)
{
    timeTextBlock.Text = position.Coordinate.Timestamp.ToString();
    sourceTextBlock.Text =
                    position.Coordinate.PositionSource.ToString();
    latTextBlock.Text = "Latitude: " +
                    position.Coordinate.Latitude.ToString();
    longTextBlock.Text = "Longitude: " +
                    position.Coordinate.Longitude.ToString();
}
```

Above you can see the code that deals with the position changed event. The `updateDisplay` method updates the display elements in the Windows Phone page. It is called from the event handler that fires when the phone detects a movement greater than the threshold that was set.

The locater status changed method will call `BeginInvoke` to perform the display update. This is because the method is running in a different context from the display system. We first saw this issue in chapter 7, where we were updating the display when a new network message arrived. I use the same technique to manage the display of changes to the status of the GPS system on the phone.

```
void locator_StatusChanged(Geolocator scodender,
                           StatusChangedEventArgs args)
{
    Dispatcher.BeginInvoke(() =>
    {
        updateStatus(args.Status);
    });
}

private void updateStatus(PositionStatus status)
{
    statusTextBlock.Text = status.ToString();
}
```

When the status of the GPS system in the phone changes it fires an event which runs a method to update the status display.

The solution in *Demo 03 Track Me* contains a program that will display the location of the phone and update the display when the phone moves more than 20 meters.

Tracking location under the Phone Lock Screen

An application can use these events to track the position of the phone but the tracking will stop if the phone lock screen appears. It is possible for an application to continue to run under the lock screen:

```
PhoneApplicationService.Current.ApplicationIdleDetectionMode =
                                IdleDetectionMode.Disabled;
```

This means that your application will not be stopped when the lock screen appears. Of course this could have terrible implications for battery life. Using the GPS system will consume quite a lot of power. If you intend to use this technique you should make the user aware of the implications and allow them an option to disable the feature.

A program can get notifications of when the user brings up and clears the lock screen by connecting event handlers to the `Obscured` and `Unobscured` events that are generated by the `RootFrame` in an application.

```
RootFrame.Obscured += RootFrame_Obscured;
RootFrame.Unloaded += RootFrame_Unloaded;
```

Note that the `RootFrame` object is declared in the `App.xaml.cs` source file.

Using a background task to track location

Although an application can run under the lock screen and track location it will stop tracking as soon as the user of the phone starts another program. If you want a program to track location all the time you will have to use a background task to do this. We will look at these in chapter 13.

10.2 Using the Map component

Now that we can obtain our position in a program, the next thing to do is to display this on a map.

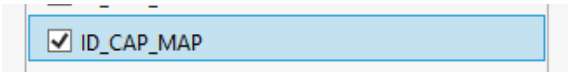


Figure 10-4 Enabling the map capability

The XAML Map control is used to display maps in an application. We can find it in the ToolBox in the designer in Visual Studio and drag it onto the page we are editing.

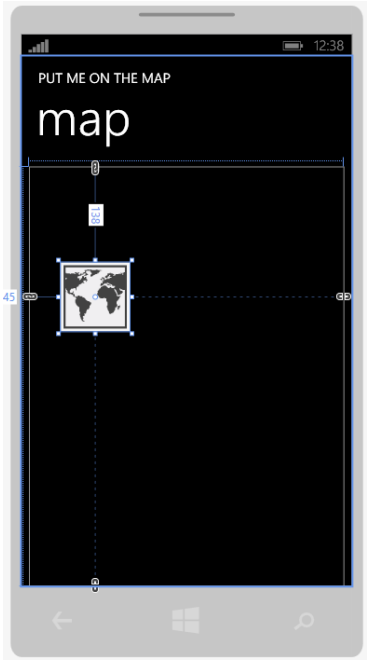


Figure 10-5 Positioning a Map

Figure 10-5 shows a map after it has been dropped onto the edit surface in Visual Studio. You can adjust the size and position of the map control on the screen by dragging the handles. Initially the map element has no name, but we can set one by using the properties window for the control.

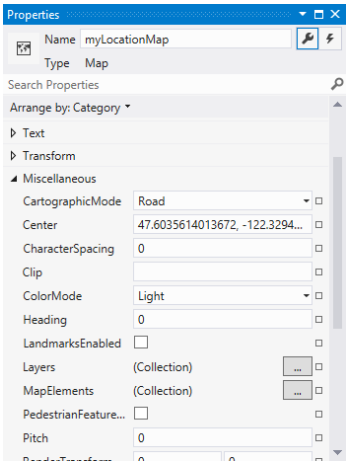


Figure 10-6 Setting the name of the map component

In Figure 10-6 you can see how I have set the name of the component to `myLocationMap`. You can also see some of the more interesting properties for this powerful control.

Of course, like any other XAML component the whole command is defined as a lump of text.

```
<maps:Map x:Name="myLocationMap" HorizontalAlignment="Left"
Margin="0,10,0,0" VerticalAlignment="Top" Width="456"
Height="587" />
```

Note that you could add a map to a page just by pasting the above code into the page you are using, but when you drag the component from the toolbox onto the page the system also sets up some extra references to libraries in your page, and so I'd advise you to use the Toolbox to add the map. You can always fine tune the XAML text by hand later.

Drawing a map centred on a particular location is very easy. The program just has to set the `Center` property of the map control.

```
private void updateDisplay(Geoposition position)
{
    GeoCoordinate drawCoordinate =
            new GeoCoordinate(position.Coordinate.Latitude,
                              position.Coordinate.Longitude);
    myLocationMap.Center = drawCoordinate;
    myLocationMap.ZoomLevel = 13;
}
```

The above version of the `updateDisplay` method will center the map on the position received from the GPS system. This can be used in the `TrackMe` application to display the location on a map, rather than as coordinates.

The method extracts the `Latitude` and `Longitude` from the position data and uses these values to build a `GeoCoordinate` value called `drawCoordinate`. The `Center` of the map is then set to this coordinate. So that you can see more easily where you are the program also sets the zoom level of the map display.

At this point I'm afraid I must mention something which you might find confusing. However it only seems fair to do this. You may be wondering why we can't just put the position value generated by GPS directly into the Center property for the map control. After all, the position value is a `Geocoordinate`, isn't it? Indeed it is, and if you look closely at the type names you'll figure out why it won't work.

The value that is generated by the GPS system is of type `Geocoordinate`. The value that is required by the map is of type `GeoCoordinate`. They are two different classes that do pretty much the same job. But they are different because the GPS coordinate has extra information in it, for example satellite information and timestamps, which a map doesn't need. Hence the two separate types. Beware of this issue, it can lead to a lot of head scratching when you can't understand why the compiler is rejecting code which looks perfect to you. I speak from experience.

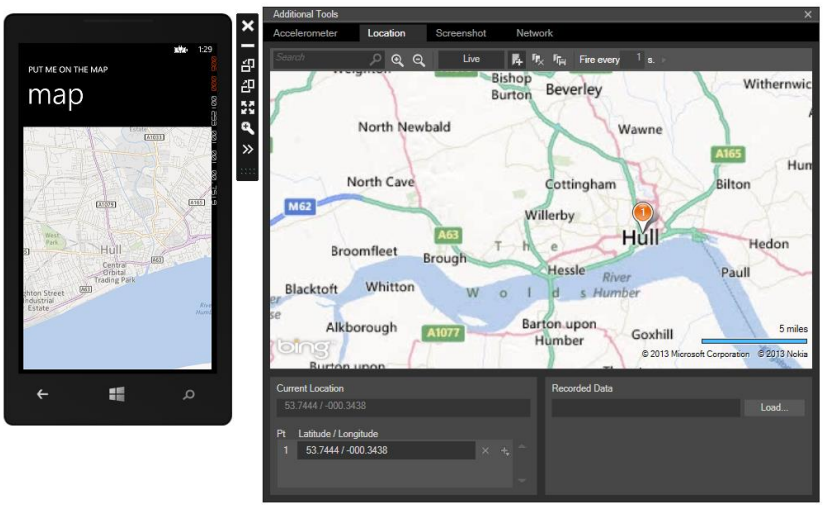


Figure 10-7 Displaying the map in the emulator

Figure 10-7 shows how the control works. I've positioned the emulator at the best city in England and the phone is now showing a map of that area. The map data is downloaded from the internet, and so for this program to work it will need network access.

Note that there is no "you are here" indication. You can add one of those by drawing a circle in the middle of the page, but this will be wrong if the user drags the map around the screen, which the control will let them do. What you really want is to be able to place a "pushpin" on the map at the required location. Unfortunately the map control doesn't allow you to do this, but the Windows Phone Toolkit (which you really should investigate) provides all this and a lot more.

The solution in *Demo 04 Put Me On the Map* contains a program that will display the location of the phone on a map and update the display when the phone moves more than 20 meters.

There are many more mapping facilities available to developers than we have space to explore here. It is very easy to use create and display routes so that an application can show the route to your shop, for example.

What We Have Learned

1. A Windows Phone device can determine the position of the phone using GPS, cellular data and Wi-Fi data.
2. A program can perform an asynchronous request to determine the position of the phone.
3. The position information can be in high precision or use a lower resolution "default" mode.
4. The emulator allows you to position the emulated phone anywhere on the world map. It is also possible to create and replay "routes" which will move the emulator phone between particular locations.
5. The GPS system can generate program events when the GPS system changes state or the phone is moved more than a particular distance.
6. Programs can bind to these events and use them to update the display or track the location of the phone.
7. If an application is replaced by another, or the lock screen is displayed, it will no longer be able to track location. It is possible for an application to allow itself to run "underneath" the lock screen, but this has implications for phone battery life.
8. If a program needs to track the location of the phone at all times it will need to create a background application to do this, of which more in chapter 11.
9. The map control allows you to display maps on the phone. The center property of the control allows you to center the map display anywhere in the world.

11 Using Bluetooth and Near Field Communications

Windows Phone devices are able to communicate with devices over short distances using Bluetooth and Near Field Communications (NFC). All Windows Phone devices are fitted with Bluetooth, but not all have NFC support. These two networking protocols are different from the ones we have seen before, in that they are usually used when the parties who want to link their devices are in the same room.

In the case of NFC the devices have to be in very close proximity, in the case of Bluetooth they can be farther apart. In this chapter we are going to find out how to set up and use Bluetooth and NFC connections and discover what they can be used for.

11.1 Using Bluetooth

The Bluetooth standard has been around for quite a while and it is a popular way to connect devices together. Bluetooth connections operate over a fairly short range, up to 10 meters, and can transfer data at a rate of up to 2.1 Mbps. Two devices that wish to communicate over Bluetooth must undertake a "pairing" process that makes each device formally aware of the other. A device that supports Bluetooth can expose a number of services for other devices to use. The services can range from keyboard, mouse, and headset to general purpose serial communication to a range of different devices.

Windows Phone 8 has support for common devices built in, and these devices are not accessible from programs that we write. However, we can use Bluetooth to transfer data from one Windows Phone to another, and we can also use Bluetooth to connect to other hardware, for example Gadgeteer devices.

We are going to create a simple intercom system that can be used on two Windows Phone devices to allow the users to send text messages between the phones. The principles are the same for any data that you may wish to transfer.

Note: There is no Bluetooth support in the Windows Phone emulator. If you want to try these sample programs you will need two Windows Phone 8 devices.

The

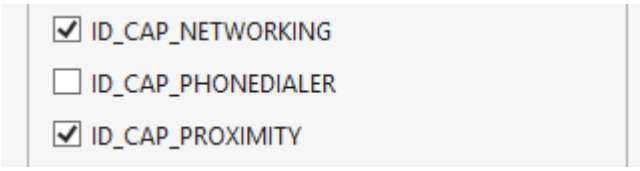


Figure 11-1 Enabling capabilities for Bluetooth connections

If you want to use Bluetooth in your application you need to enable the Networking and Proximity capabilities, as show in Figure 11-1 above. A program should also include the relevant namespaces

```
using Windows.Networking.Proximity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;
```

This provides access to the Bluetooth networking classes and also the standard networking, which runs on top of Bluetooth. The good thing about this way of working is that we will see a lot of familiar code when we start assembling and sending messages.

11.2 The Intercom Program

The simplest way to use Bluetooth in a Windows Phone 8 application is for an application on one device to talk to the same application on the other device. This is what we are going to do. This works well for multi-player games, as well as for the chat application that we are going to create. Setting up a connection and transferring data is achieved by going through a series of steps which we can explore by creating a simple intercom application. This might seem a bit daunting, but each step has a particular purpose and if you read through the sequence carefully you will find out how you can use data communications in your programs.

The Intercom program will have three pages. On the first page the user will be asked to enter their intercom name. On the second they will search for someone who wants to talk to them and on the third page they will actually have their conversation.

The Start page

Figure 11-2 The start page

On the start page the user enters their name and presses the button to join in the conversation.

```
private void joinInButton_Click(object sender, RoutedEventArgs e)
{
    if (nameTextBox.Text == "")
    {
        MessageBox.Show("Please enter your name to chat");
        return;
    }

    // Store the chat name in the app
    App thisApp = Application.Current as App;

    thisApp.ChatName = nameTextBox.Text;

    // Navigate to the page to find a chat partner

    NavigationService.Navigate(new Uri("/FindPage.xaml",
                               UriKind.Relative));
}
```

The join button stores the name of this user in the application class, `App.xaml.cs`. This is so that the search page code can load the name and use it when starting the Bluetooth connection.

The Search page

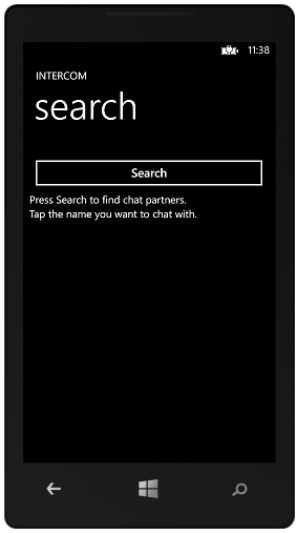


Figure 11-3 The search page

When the application is on the search page the user can search for people to talk to. At the same time the application is also broadcasting the hostname so that other stations can find and connect with the application.

The code that sets up these behaviours is in the `onNavigatedTo` method in the `SearchPage` class:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    App thisApp = Application.Current as App;

    // Set up the PeerFinder
    PeerFinder.DisplayName = thisApp.ChatName;
    PeerFinder.ConnectionRequested +=
                            PeerFinder_ConnectionRequested;
    PeerFinder.Start();

    // Create a collection of peers that we have searched for
    peerInfo = new ObservableCollection<PeerWrapper>();

    // Bind the collection to a display element
    peersListBox.ItemsSource = peerInfo;
}
```

The first part of the method configures the `PeerFinder` class and starts the device broadcasting its availability. The `PeerFinder` class is `static`, in that we don't need to create an instance of the class, it is built into the system. It looks for "peer applications", in other words it will only find copies of the same application running on another machine.

The code above sets the `DisplayName` property of the `PeerFinder` to the name that the user typed in at the Start page. This is the name that other users will see when the look for you. It then connects a method to the `ConnectionRequested` event that `PeerFinder` can generate. If a user on another device who is running this application attempts to connect with this one, the event is fired so that the user can be asked if they would like to chat with that user.

The second part of the method deals with the display of the list of potential hosts that the user might like to chat with.

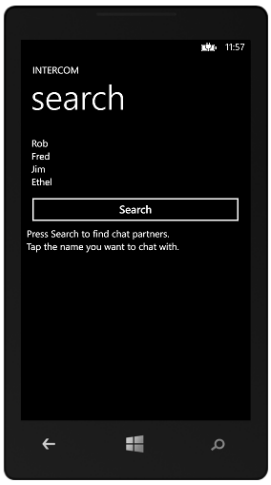


Figure 11-4 The search page with a list of hosts

Figure 11-4 shows how this will work. When the user taps the Search button the program will search for all the hosts available for a conversation (we will see how this is achieved next). If there are lots of people out there who want to talk with you (and why not, I hear you are a very popular person) then these hosts are displayed as a list. We have seen how to do this in chapter 4. We can build up a collection of elements and then set this list as the `ItemsSource` property of a `ListBox`. The last two statements in the method do just this. The first one creates a new `ObservableCollection` of `PeerWrapper` items and the second one assigns this list to the `peersListBox` on the screen. Of course, the list is initially empty, and so the program display looks like the one in Figure 11-3. It is only when some hosts have been found and the collection given some elements that we actually see some names there.

The `PeerWrapper` class is something we will use manage the display of peer information, we will come to how this is used in a little while.

Once the program has displayed this page it is waiting for two things to happen. Either the user of the program will press the Search button, to see if there is anyone out there who would like to talk to them, or another host will attempt to connect.

Searching for Hosts

The event handler for the Search button will search for all the peers (copies of the application) that are available for a conversation.

```
async void showPeers()
{
    try
    {
        var peers = await PeerFinder.FindAllPeersAsync();

        // Clear the list of names
        peerInfo.Clear();

        if (peers.Count > 0)
        {
            // Add peers to list
            foreach (var peer in peers)
            {
                peerInfo.Add(new PeerWrapper(peer));
            }
        }
    }
```

```
        catch (Exception ex)
        {
            if ((uint)ex.HResult == ERR_BLUETOOTH_OFF)
            {
                MessageBox.Show("Bluetooth turned off");
                ConnectionSettingsTask connectionSettingsTask =
                                new ConnectionSettingsTask();
                connectionSettingsTask.ConnectionSettingsType =
                                ConnectionSettingsType.Bluetooth;
                connectionSettingsTask.Show();
            }
            else if ((uint)ex.HResult == ERR_MISSING_CAPS)
            {
                MessageBox.Show("No Bluetooth capability");
            }
            else if ((uint)ex.HResult == ERR_NOT_ADVERTISING)
            {
                MessageBox.Show("Not advertising host");
            }
            else
            {
                MessageBox.Show(ex.Message);
            }
        }
    }
```

This method looks a bit scary, but actually it is quite simple. The hard work is done in the first few lines, and the rest of the method is just error handling. This is how most programs end up. We can work our way through it, one section at a time.

```
var peers = await PeerFinder.FindAllPeersAsync();
// Clear the list of names
peerInfo.Clear();

if (peers.Count > 0)
{
    // Add peers to list
    foreach (var peer in peers)
    {
        peerInfo.Add(new PeerWrapper(peer));
    }
}
```

This part of the method is where the peers are actually discovered. The `FindAllPeersAsync` method searches for any hosts and returns them. Note that it is called using `await`. This is because, as the name implies, it runs asynchronously.

This method actually returns a result, whereas some asynchronous methods just perform a task. The variable `peers` is set to this result. Note that this variable has the type `var`. Using the type `var` means that we are saying "I don't want to go into full details of exactly what type this variable is, but I'd like to just use it and have the compiler pick me up if I make any mistakes". It is a form of collection of peer items, and as long as we treat the items correctly the program will work fine.

The first thing the code does with the peers that came back is to check to make sure that there are actually some peers in the list. It can use the Count property of the result to determine how many results came back. If the program received more than one item it will work through the list. For each peer in the list (note the use of `var` again) it creates a new `PeerWrapper` from that peer. At this point we need to take a look at the `PeerWrapper` class:

```
public class PeerWrapper
{
    public PeerWrapper(PeerInformation peerInformation)
    {
        this.PeerInfo = peerInformation;
    }

    public PeerInformation PeerInfo { get; set; }

    public override string ToString()
    {
        return PeerInfo.DisplayName;
    }
}
```

This class is a wrapper round a `PeerInformation` value. It information about one of the peers that was returned by the call to `FindAllPeersAsync` and provides a `ToString` behaviour which gives the name of the `PeerInformation` that it holds. The idea is that when I connect a collection of these things to the `ListBox` that is going to display them, the `ListBox` will use the `ToString` behaviour of each item in the list to get what appears on the screen. This will be the name of the item.

If this is a bit hard to understand, remember what I'm trying to do. I need to display a list of `PeerInformation` values for the user. The thing I want from each of the values is their `DisplayName`. So I make a class that holds a value and then returns the name if you ever ask it for its string rendition. You can think of this as a primitive kind of View Model class if you like.

So when the `foreach` loop has finished going through all the peers that have been discovered it will have built a list of items that can be displayed in the list. Because this is an `ObservableCollection` the display will be updated automatically to show the hosts that are available.

The rest of the method deals with any errors that may be produced by the call of `FindAllPeersAsync`. These take the form of exceptions that may be thrown. The code has an appropriate behaviour for each one of them.

```
catch (Exception ex)
{
    if ((uint)ex.HResult == ERR_BLUETOOTH_OFF)
    {
        MessageBox.Show("Bluetooth turned off");
        ConnectionSettingsTask connectionSettingsTask =
                        new ConnectionSettingsTask();
        connectionSettingsTask.ConnectionSettingsType =
                        ConnectionSettingsType.Bluetooth;
        connectionSettingsTask.Show();
    }
    else if ((uint)ex.HResult == ERR_MISSING_CAPS)
    {
        MessageBox.Show("No Bluetooth capability");
    }
    else if ((uint)ex.HResult == ERR_NOT_ADVERTISING)
    {
        MessageBox.Show("Not advertising host");
    }
    else
    {
        MessageBox.Show(ex.Message);
    }
}
```

The error values are declared as constants earlier in the program.

Opening the Bluetooth settings page

The only one of interest is the behaviour that is triggered if an exception is triggered because Bluetooth is not turned on. This launches a new `ConnectionSettingsTask` and uses it to display the Bluetooth menu from the settings on the phone. This shows how easy it is to display system menus.

Figure 11-5 Bluetooth settings

The user can turn Bluetooth on and then re-run the program. Note that because of the "user is always in charge" ethos of Windows Phone it is not possible for the application to turn Bluetooth on automatically. This is a deliberate design decision by the Windows Phone developers. We will find out more about Launchers and Choosers in chapter 12 section 2.

Selecting a host to connect with

Now that we know how the program displays a list of hosts, the next thing that we need to consider is how the user selects one of them to connect with.

The selection is made by the user selecting one of the hosts in the list. When this happens an event handler runs in the program.

```
private void peersListBox_SelectionChanged(object sender,
                                           SelectionChangedEventArgs e)
{
    if (peersListBox.SelectedItem != null)
    {
        // Get the selected peer
        PeerWrapper selectedPeerInfo =
                    peersListBox.SelectedItem as PeerWrapper;

        startChat(selectedPeerInfo.PeerInfo);
    }
}
```

We have seen how we can connect handlers to events that are produced by user actions on the display, this method runs when the selected item in the hosts list is changed. The method first makes sure that there actually is a selected item and then, if there is on, it creates a `PeerWrapper` value from the selected item and then calls the `startChat` method to start a chat with that particular peer.

```
void startChat(PeerInformation peer)
{
    // Store the peer details in the app
    App thisApp = Application.Current as App;

    thisApp.ActivePeer = peer;

    // Navigate to the chat page

    NavigationService.Navigate(new Uri("/TalkPage.xaml",
                               UriKind.Relative));
}
```

The `startChat` method doesn't do much, it just saves the peer to be used in the application and then navigates to the `TalkPage`, which will set up the conversation.

Responding to Chat Requests

As well as a user initiating a chat, they can also receive chat requests from other devices. If this happens the `PeerFinder_ConnectionRequested` method is called.

```
void PeerFinder_ConnectionRequested(object sender, ConnectionRequestedEventArgs args)
{
    try
    {
        this.Dispatcher.BeginInvoke(() =>
        {
            // Ask the user if they want to accept the incoming
            // chat request.
            var result =
               MessageBox.Show(args.PeerInformation.DisplayName,
               "Do you want to chat?", MessageBoxButton.OKCancel);
            if (result == MessageBoxResult.OK)
            {
                startChat(args.PeerInformation);
            }
        });
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Because the method is running on an event from the `PeerFinder` and wishes to use the display to show a message box, it uses the `Dispatcher` from the form to display the question and receive the answer. If the user wants to start a conversation the method simply calls the same `startChat` method we saw above. This navigates to the final page in our program, where the conversations take place.

Before the program leaves this page it is best to remove the event hander for new chat requests. This is just in case anyone tries to start a conversation with us when we are not expecting it. We can do this by overriding the `OnNavigatedFrom` method:

```
protected override void OnNavigatingFrom(
            System.Windows.Navigation.NavigatingCancelEventArgs e)
{

    PeerFinder.ConnectionRequested -=
                                    PeerFinder_ConnectionRequested;
    base.OnNavigatingFrom(e);
}
```

The Talk Page

Figure 11-6 Having a conversation

The chat page provides a textbox which shows the conversation and a message entry dialogue that is used to enter and send messages. The conversation itself is set up when the page is loaded.

```
// Network connection items
StreamSocket socket;
DataReader dataReader;
DataWriter dataWriter;
```

These are the network connection items that need to be set up. The interesting thing here is that these are exactly the same items that we would use if we were making a connection using the internet. In fact we saw them first in Chapter 8, where we discussed networking. The `StreamSocket` provides the raw data transfer, and the `DataReader` and `DataWriter`. When the page loads these must be set up:

```
async protected override void OnNavigatedTo(NavigationEventArgs e)
{
    // Get the peer
    App thisApp = Application.Current as App;

    try {
        socket = await PeerFinder.ConnectAsync(thisApp.ActivePeer);

        // We can save battery by not advertising our presence.
        PeerFinder.Stop();

        peerNameTextBox.Text = thisApp.ActivePeer.DisplayName;

        // Start a thread running to process incoming messages
        Thread readThread =
              new Thread(new ThreadStart(ProcessIncomingMessages));

        readThread.Start();
    }
    catch (Exception ex) {
        // Show the message and return to the search page
        MessageBox.Show(ex.Message);
        closeConnections();
        returnToSearch();
    }
}
```

The method creates the socket connection, based on the `PeerInformation` that was stored in the `App` class. It then starts off a thread to process incoming messages. These will be displayed on the screen. If anything goes wrong the program displays the exception details, closes any connections that may be open and then returns to the search page. We will look at how these methods work a little later in the chapter.

Receiving Messages

The talk page can receive messages at any time. The `processIncomingMessages` method has the job of reading each message and displaying it. We are using a simple protocol that when the system receives an empty message, that indicates that the conversation is over and that the program should return to the search page.

```
async void processIncomingMessages()
{
    bool finished = false;

    while (!finished)
    {
        try
        {
            var message = await getMessage();
            if (message == "")
                finished = true;
            // Add to chat
            displayReceivedText(message);
        }
        catch (Exception)
        {
            finished = true;
        }
    }
    closeConnections();
    returnToSearch();
}
```

The method runs until an empty message is received or an exception is thrown. The `getMessage` method returns a string that has been sent from the remote terminal.

```
async Task<string> getMessage()
{
    if (dataReader == null)
        dataReader = new DataReader(socket.InputStream);

    // Each message is sent in two blocks.
    // The first is the size of the message.
    // The second is the message itself.
    await dataReader.LoadAsync(4);
    uint messageLen = (uint)dataReader.ReadInt32();
    await dataReader.LoadAsync(messageLen);
    return dataReader.ReadString(messageLen);
}
```

We have seen this way of working before in chapter 8. We know that the underlying data connection just provides a way of transferring streams of bytes between systems. The `DataReader` and `DataWriter` classes can send particular types of data over a stream. The string of text that the user types in is actually sent as two items of data, the length of the string (as an integer) and a sequence of bytes of that length. If you think about it, this is the only way it would work. We can't just send a bunch of characters as the receiver would have no idea of knowing when the end of the string had been reached. So the `GetMessage` method above first reads a `messageLen` value and then reads that number of bytes to build up the message. It then returns the string it has received. Note that the length of the message is sent as a 4 byte value encoded as a 32 bit integer.

Reading data from a stream like this can take some time to complete and so the method is implemented as an asynchronous one. Note how the return of the method is a `Task` that returns a string. When the method is called the `await` keyword will cause the calling method to be suspended until the message is ready.

You can use this technique to build up messages of your own which can contain any kind of data. Note that you must make sure that the transmitter and receiver agree on the data format, or your program will fail or at the very least do some extremely strange things.

The message is displayed in front of the user by the `displayReceivedText` method.

```
void displayReceivedText(string message)
{
    // Get the peer
    App thisApp = Application.Current as App;

    displayMessage(thisApp.ActivePeer.DisplayName + " : " +
                   message);
}
```

This method assembles a message string which includes the name of the person who sent the message. It then calls `displayMessage` to add this to the message box on the screen.

```
void displayMessage(string message)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        chatTextBlock.Text = chatTextBlock.Text + "\r\n" + message;
    });
}
```

Note that this method uses a `Dispatcher` as the message processor is not running on the same thread as the display.

Sending Messages

The user can type in a message and press the Send key to send it.

```
void sendButton_Click(object sender, RoutedEventArgs e)
{
    if (sendTextBox.Text == "")
        return;

    sendMessage(sendTextBox.Text);
}
```

This is the event handler for the send button. If there is no text to send the method returns immediately. Otherwise it calls the send Message method to actually assemble and deliver the message.

```
async void sendMessage(string message)
{
    if (dataWriter == null)
        dataWriter = new DataWriter(socket.OutputStream);

    // Each message is sent in two blocks.
    // The first is the size of the message.
    // The second is the message itself.
    dataWriter.WriteInt32(message.Length);
    await dataWriter.StoreAsync();

    dataWriter.WriteString(message);
    await dataWriter.StoreAsync();

    displaySentText(message);

    sendTextBox.Text = "";

    if (message == "")
    {
        closeConnections();
    }
}
```

The method assembles a message that will make sense to the receiver. If the messages is an empty string it closes down the connections after the message has been delivered. The `closeConnections` method disposes of any open streams and network sockets.

```
void closeConnections()
{
    if (dataReader != null)
    {
        dataReader.Dispose();
        dataReader = null;
    }

    if (dataWriter != null)
    {
        dataWriter.Dispose();
        dataWriter = null;
    }

    if (socket != null)
    {
        socket.Dispose();
        socket = null;
    }
}
```

Ending the Conversation

Either party in the conversation can end it at any time, simply by pressing the Back button on the conversation screen:

```
private void PhoneApplicationPage_BackKeyPress(object sender,
                        System.ComponentModel.CancelEventArgs e)
{
    // Send an empty string to shut down the other end
    sendMessage("");
}
```

We have seen that we can capture this event and run code when the user presses the Back button on the phone. In this case the handler sends an empty string, which will shut down the remote system and cause them to return to the search page too.

The Intercom application

The Intercom application works quite well. You can use it as the basis of any communication between two devices. The program itself asks for confirmation before setting up the connection, but there may be no need to do this. It would be very easy to create a multi-player came using this mechanism. One final thing to bear in mind is that the way sockets and DataReader/DataWriter objects have been used is exactly the same as their use over a network connection. In other words you can use all the above techniques to format messages that can be sent over other networks.

The solution in *Demo 01 Bluetooth Intercom* contains a program that will implement the intercom behaviours described above. If you do want to use this as the basis of your own networked application it would work well, but you may have to add a little more error handling.

Bluetooth and Devices

It is possible to use Bluetooth to talk to external hardware. In this case you will have to use a slightly different form of the peer finder mechanism which lets a connection target specific Bluetooth hardware profiles. If you want to use other devices you will also need to pair the devices with the phone, using the Bluetooth settings page.

11.3 Using Near Field Communications

Near Field Communications (NFC) is a short range networking mechanism. Short range in this case means just a few centimetres. You can use it to link programs in phones in the same way as other network types. The communication is based on the same socketed based infrastructure we have already seen, but the contact is initiated by the users “tapping” their phones together.

In this section we are going to focus on using NFC to talk to “tags” which are tiny devices which can be powered by the signal in the phone.

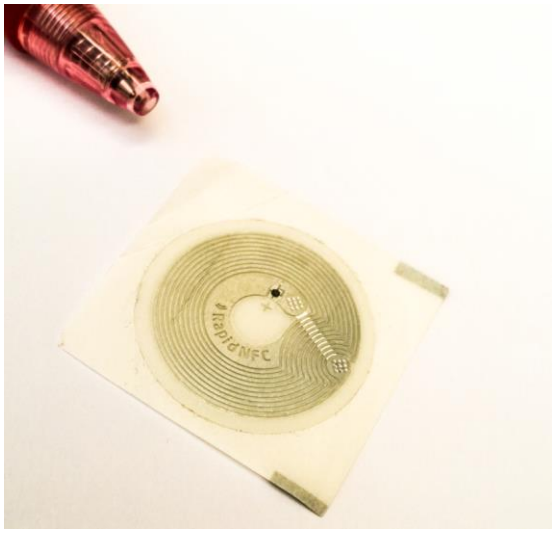


Figure 11-7An NFC tag

Figure 11-7 shows a tag and a pen to give some scale. The tag above is self-adhesive and can be peeled off and stuck on a flat surface. You can also get tags built into plastic labels and if you are feeling rich there are companies who will print custom tags for you. A single "bare bones" tag like the one above will cost around a dollar or so, with the price dropping if you buy a very large number. The chip that provides the intelligence in the tag is in the tiny dot towards the middle of the tag. The rest of the tag is made up of antenna and power coil.

NFC tags can contain small amounts of data which a phone can read and write, which leads to lots of interesting ways you can use them. A program could store a web address in a tag, or use a tag to activate a particular program behaviour. You could use them as tickets for a party, to uniquely identify your dog, or to store your favourite music track of the moment.

Data is stored in an NFC tag using the NFC Data Exchange Format (NDEF). If you buy any blank tags you should make sure they have been formatted with this standard, otherwise they won't work with the phone.

The tag itself does not contain a battery, it is powered up by the magnetic field produced by the reader in the phone.

Note that one potential issue with NFC tags and Windows Phone is that the phone is unable to "lock" a tag. This means that once you have written a tag there is nothing to stop someone else re-writing the tag with other content. There are specialised devices that you can use to lock tags but it is not possible to use a program on Windows Phone to do this.

The NFC hardware is in many, but not all, Windows Phone devices. If you want the phone to be able to interact with NFC devices you must enable the tap+send option in the Settings menu:

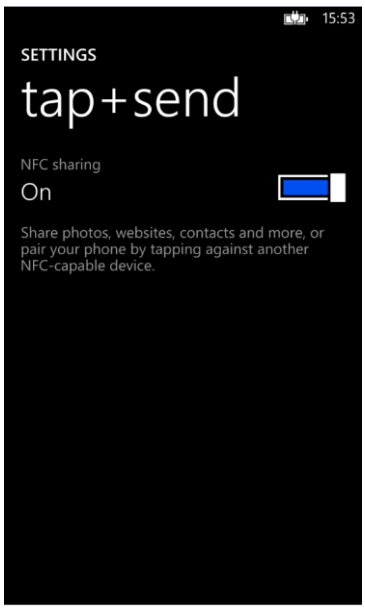


Figure 11-8 Enabling tap+send in Settings

If this option is turned off the phone will not recognise NFC devices. Turning the option on has a small effect on battery life, but in my experience it doesn't make much difference.

The NFC Linker Program

We are going to start by writing a program that will write web links onto an NFC tag. We will use the NDEF format for `WindowsUri`, which means that any Windows Phone will recognise it as a link and offer the user the opportunity to open the link if it is tapped against the phone.

Figure 11-9 Entering a url

Figure 11-9 shows the program in action. The user types in the url to be written to the NFC tag and then presses the Write button to write it.

A program that wishes to use the NFC tags must enable the Proximity capability.

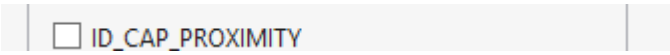


Figure 11-10 Proximity Capability

The classes that implement the NFC tag interface are in the Proximity namespace, which you should include in your programs.

```
using Windows.Networking.Proximity;
```

The hard work will be done by the `ProximityDevice` class which provides the connection to the NFC devices.

```
ProximityDevice device;

// Constructor
public MainPage()
{
    InitializeComponent();
    device = ProximityDevice.GetDefault();
}
```

The code above gets the default `ProximityDevice` for the program to use. This takes place when the XAML page for the application is constructed. Note that we never create a new instance of this class, instead we get access to one that is built into the system.

When the user has entered a url that is to be written to the tag they can press the Write button to send the text into the tag.

```
private void writeButtonClicked(object sender, RoutedEventArgs e)
{
    if (urlTextBox.Text.Length == 0)
        return;

    statusTextBlock.Text = "Tap the tag";

    IBuffer linkBuffer =
        Encoding.Unicode.GetBytes(urlTextBox.Text).AsBuffer();

    device.PublishBinaryMessage("WindowsURI:WriteTag", linkBuffer,
                                LinkSavedCallback);
}
```

The clicked event handler first makes sure that there is a link to write. Then it sets a status block to tell the user that they must tap the NFC tag to initiate the data transfer. Next it creates a buffer, called `linkBuffer` that will hold the data to be transferred.

The statement that sets the `linkBuffer` value looks a lot like magic, but it is not too tricky. The `Encoding` class lives in the `System.Text` namespace and contains lots of helpful classes and methods that convert text to and from various kinds of encoding. It contains a Unicode class that will convert a string of text into a buffer full of bytes. Bytes are the eight bit values that we can send to the card. Unicode is a mapping of text onto values. The idea is that the program will write eight bit byte values onto the tag, and something has to convert the string of text into those eight bit values. That is what the `GetBytes` method does.

The part of this statement that is the most confusing is the `AsBuffer` element at the end of the statement. This takes the array of bytes and converts it into an object that implements the `IBuffer` interface. Lots of network commands expect data in this format.

`AsBuffer` is in the `System.Runtime.InteropServices.WindowsRuntime` namespace and is very useful for preparing data for use on the network.

The final statement in the method above asks the device to publish the link to the card. It does this by calling the `PublishBinaryMessage` method. This method has three parameters. The first identifies the format of the message to be created, the second is the message text itself and the third parameters is the name of a "callback" method that is called when the tag has been written.

When the message has been written to the tag the `LinkSavedCallback` method will be called.

```
private void LinkSavedCallback(ProximityDevice sender,
                               long messageId)
{
    sender.StopPublishingMessage(messageId);

    Dispatcher.BeginInvoke(() =>
    {
        statusTextBlock.Text = "Tag written";
    });
}
```

This method tells the proximity device to stop sending that message and then updates the status method to show that the message has been delivered.

The tag will retain the url and if a Windows Phone user taps the tag it will ask if they would like to visit the site on the tag.

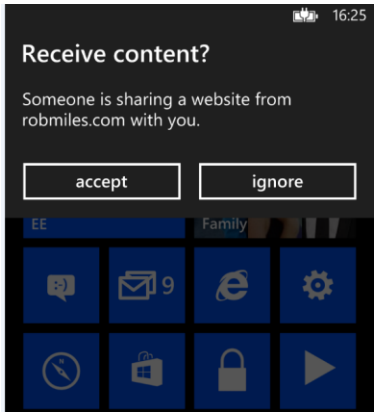


Figure 11-11Confirming navigation to a url

Note that the user is shown the url to be visited, and they are also asked to confirm the action. This is another demonstration of the way that security is paramount on Windows Phone. There is no way that a tag can be created that will automatically navigate to a target. This is to prevent phones being hijacked by NFC tags and directed to dangerous parts of the web.

The solution in *Demo 02 NFC Linker* contains a program that will write a link onto a tag.

What We Have Learned

1. Bluetooth is a short range networking technology with is built into Windows Phone and can be used to connect devices and exchange data.
2. The Bluetooth hardware in a Windows Phone can be used to connect a variety of devices to the phone. Each kind of Bluetooth device is characterised by a particular device profile.
3. A Windows Phone is paired with the device with which it is going to communicate.
4. Two copies of the same application, running in different phones can connect directly without explicitly pairing by the users.
5. A program can find a list of "peer" applications and create a connection to one of them.
6. The Bluetooth data connection between two peer applications is based on a StreamSocket and from a programming point of view can be used in exactly the same way as networking sockets.
7. Bluetooth can also be used to connect a Windows Phone with other external devices, but these must be paired with the phone before they can be used.
8. Near Field Communications (NFC) abilities built in to some Windows Phones provide broadly the same communication facilities as Bluetooth as far as Windows Phone applications are concerned. Programs can connect and share data, although the range is much more restricted (only a few centimetres) and the data rate is reduced. Normally a connection is initiated by tapping one phone against the other.
9. The NFC support extends to connection with passive NFC "tags" which are powered by the phone and can store small amounts of data, for example links to programs or web pages.
10. The tags are formatted using NFC Data Exchange Format (NDEF) which specifies different types of data. Windows Phone devices are not able to "write protect" the contents of a tag.

12 How Applications Run

In this chapter we are going to explore how programs actually run on Windows Phone, and the ways that you can manage their execution.

12.1 Fast Application Switching

The Windows Phone is a very powerful device. Computers of this power filled entire rooms just a few years ago. However, when compared with desktop systems it is somewhat constrained. It does not have a processor which is quite as powerful, and it is restricted by the limited storage capacity of the battery. While the phone can do a lot of processing, it is best if this is kept to a minimum to make battery life as long as possible.

Now we are going to find out how an application can be made to provide a good user experience within the single tasking environment provided on the Windows Phone. This provides a good insight into how programmers often have to make use of systems that are compromised by underlying limitations of the platform and operating system.

Task Navigation in Windows Phone

The Windows Phone operating system is inherently multi-tasking. This means that it can easily run more than one process at the same time. This is how it can play music while you use the phone to check your email. However, for a number of very sound technical reasons this multi-tasking ability is not extended to applications running on the system. The creators of the Windows Phone system took a deliberate design decision not to allow more than one application to be active at any given time.

In some ways this is a good thing. It stops our programs from looking bad just because another application running in the phone is stealing all the memory or processor time. But it does give us some programming challenges that we must address when we write Windows Phone applications.

When we write an application the aim should be to make it appear that the application never stops when a user leaves it and goes off to do something else. This "leaving and going off to do something else" is a behaviour that is strongly encouraged by the design of the Windows Phone user interface. We know how an application can store data using the mass storage on the phone, what we have to do is store and retrieve data to give the illusion that our program was never paused.

Later on we will see how we can create applications that do perform some background processing on behalf of the user. However, we will find that this ability is carefully managed to make sure that anything our program does in the background does not compromise the experience that the user gets.

The Start and Back buttons

While Windows Phone does not have multi-tasking it does have features which have been designed to make it easy for a user to move in and out of programs. These are based on the hardware **Back** and **Start** buttons that are fitted to every Windows phone device.

Put simply, **Start** moves the user forward to a new program that they may wish to run and **Back** takes the user back to a previous one. A user can press **Start** at any time to open the Start screen and select a different program to run. When they have finished using that different program they can use **Back** to return to the original one.

Used in combination these buttons greatly enhance the user experience. It is easy to press **Start** to send a quick SMS message and then press **Back** to return to whatever you were doing before. The **Back** and **Start** buttons allow the user to navigate through a stack of recently used applications in a similar way to browsing the internet, clicking on links and using the Back command to return to the previous page.

Pressing the **Start** button does not cause an application to be removed from memory to make way for a new one; instead it is "deactivated". We can regard a deactivated application as "waiting in the wings", ready to be called back onto the stage at some later time. Of course the call back to the stage may never come (just for real life performers) and so when the application is deactivated it must store any important data just in case it the user never returns to it.

The Back Button Long Press

In addition to pressing and releasing the **Back** button a user of the phone can hold it down for half a second or so to activate a display of applications that are presently "waiting in the wings". If the user selects one of these applications it is restarted exactly as if the user had returned to the application via the Back button.

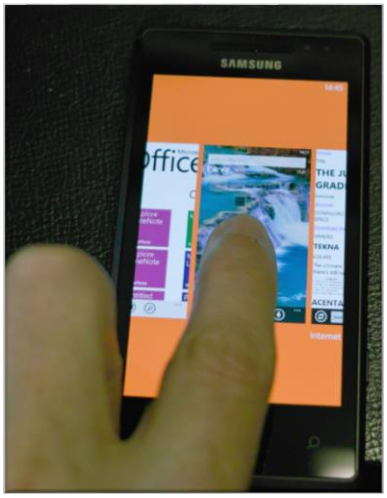


Figure 12-1 Selecting from active applications

Figure 12-1 shows how waiting applications are selected. I am panning between the currently active applications (these include Office and the Bing search screen) to select the one that I want to go back to.

Creating a "Well Behaved" Application

The result of all this is that an application must be made adept at saving its state and then restoring itself to the same state when the user returns to it. The Windows Phone system helps us do this by sending an application messages that indicate what is going to happen to it. Next we are going to explore the situations in which the messages are sent and how we can make our applications and games respond correctly to give the appearance that they are always active.

Understanding Fast Application Switching

The name for this set of behaviours is "Fast Application Switching". Because deactivated programs are usually held in memory the process of switching between them should be very quick. If we are going to understand how to make programs that behave properly we are going to have to learn some of the terminology used by the Windows Phone designers to describe what happens to make this work.

The Fast Application Switching Event Methods

The `App.xaml.cs` file in a Windows Phone application contains methods that are called when the various fast application switching events occur:

```
// Code to execute when the application is launching (eg, from Start)
// This code will not execute when the application is reactivated
private void Application_Launching(object sender, LaunchingEventArgs e)
{
}

// Code to execute when the application is activated (brought to foreground)
// This code will not execute when the application is first launched
private void Application_Activated(object sender, ActivatedEventArgs e)
{
}

// Code to execute when the application is deactivated (sent to background)
// This code will not execute when the application is closing
private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
}

// Code to execute when the application is closing (eg, user hit Back)
// This code will not execute when the application is deactivated
private void Application_Closing(object sender, ClosingEventArgs e)
{
}
```

We first saw the file `App.xaml.cs` in section 4.7 *Pages and Navigation* when we used it as a place to create variables that can be shared by all the pages in an application. It is the class that controls the creation and management of the XAML pages. Our job is to make our applications use these events to give the user the impression that our program is always ready for use.

The Windows Phone Application Lifecycle

Now that we know what event messages there are, and how the user moves from one application to another, we can consider how the messages work in the lifecycle of an application.

Program Launching

A user launches a program by selecting it from those available on the phone. When a program is launched the program files are loaded into memory and executed by the Windows Phone processor. During the launch process the `Application_Launching` method is called

```
private void Application_Launching(object sender,
                                   LaunchingEventArgs e)
{
}
```

This method should load any data from isolated storage. However, if there is a large amount of data to be read it might be best to load this in a separate thread (i.e. in the background). This is because the application will not display anything on the screen until this method has finished running. If the program spends a long time loading data this means your program will seem very slow to load.

There are two things that the user can do next. The user can exit the program by pressing the Back button at the top level menu of the program or the user can move to another program by pressing the Start button.

Pressing Back to Close an Application

Windows Phone applications do not have an “exit” option that can be used to close them. However, this does not mean that it is impossible to close an application. If the user presses the Back button at the very “top level” of the application (i.e. the screen that is displayed when the application was started) the application will be closed at this point. When the application closes it is removed from memory and the previous application (if there is one) is resumed. This gives the behaviour where a user can “back” out of an application and return to the one they interrupted.

Just like a Windows PC application, a Windows Phone application uses the closing event so that it can save any open files.

```
private void Application_Closing(object sender, ClosingEventArgs e)
{
}
```

It is even possible to allow the user the chance to confirm that the program is being closed. The Microsoft Word application on the phone will display a "Do you want to save your text" message if you try to back out of it after you have entered some text.

Unfortunately for us users will not always close a program "cleanly" like this. They are much more likely to press the Start button and go off and do something else rather than close one application to run another. When we design a Windows Phone application we need to take care to make sure that we give proper consideration to how the program will behave when the user leaves it and returns to it. We must make sure that the application always does what the user expects, and does not suddenly discard changes or lose data.

Deactivating an Application by pressing Start

When the user presses Start the currently active program is deactivated. This means it is initially made "dormant". Dormant programs are still in memory, but are not presently active. They are ready to be resumed using "Fast Application Switching".

```
private void Application_Deactivated(object sender,
                                     DeactivatedEventArgs e)
{
}
```

At the point that an application is being made "dormant" it is sent a "deactivated" message to indicate that it is being "moved to the wings" to make way for another performer on the stage. This is the point at which the application must save the state of the current interaction and also save data in isolated storage in case it is not resumed. A program can also be made "dormant" if the phone lock screen is displayed because the phone has not been used and has timed out. A program may also be made dormant if the user presses the Search button, or starts using the camera to take a photograph.

Converting a Dormant Application into a Tombstoned one

A real theatre would have problems if a large number of performers were sent to the wings, the wings might get very crowded. If this happened some of the performers would be sent back to their dressing rooms. If these performers are ever needed again they will need to be called back, which means it will take longer for them to resume their performance.

Exactly the same thing happens in Windows Phone. As more and more applications are deactivated and made dormant the memory in the phone will eventually fill up. At this point some of the applications need to be removed from memory to make room for others. In Windows Phone terms they are converted from "dormant" (in memory and available for Fast Application Switching) to "tombstoned" (removed from memory). The user can still resume a tombstoned application if they press Back enough times, or select it from the "Long Press" menu but it will take slightly longer to reactivate.

If a tombstoned program is ever reactivated it will be loaded into memory and all the objects in the program will need to be constructed from scratch, just as they would be if the program was loaded for the first time.

At this point we might be forgiven for thinking that being tombstoned is just like being exited. However, this is not quite the case. The great thing about a tombstone is that we can write things on it. The Windows Phone system provides some space in memory where an application can store state information if it is ever deactivated. This information can then be loaded back into the application if it is ever reactivated. The information is stored when the application is deactivated and maintained whether the application is made dormant or tombstoned.

The Windows Phone system also uses this state memory. If you deactivate an application at a particular page you will find that if you reactivate it you are returned to the same page. The Windows Phone system uses the state storage in memory to record the active page when it is deactivated.

Resuming a Dormant or Tombstoned Application

When an application is being started it will receive either a "Launching" message if it is being launched as new or an "Activated" message if it is being reactivated.

The application can then test to see if it is being reactivated from being "dormant", or if it is being reactivated from being "tombstoned".

```
private void Application_Activated(object sender,
                                   ActivatedEventArgs e)
{

    if (e.IsApplicationInstancePreserved)
    {
        // Dormant - objects in memory intact
    }
    else
    {
        // Tombstoned - need to reload our objects
    }
}
```

This method runs when the application is activated and tests which form of activation is being performed. When an application is restarted from the dormant state it will have all its memory objects intact and can just resume running. When an application is restarted from the tombstoned state it must reload all its data as it will have been removed from memory.

Note that an application does not get a message when it is moved from the dormant to the tombstoned state, it can only tell which state it was in if it is ever restarted.

Fast Application Switching in an application

We are now going to take a look at an application and consider how we would make it into one which works correctly with respect to Fast Application Switching. In Chapter 5 we created an application called "JotPad" that allowed the user to store text jottings on the phone. To investigate Fast Application Switching we are going to use an enhanced version of JotPad called "The Captain's Log".

An Introduction to "The Captain's Log"

Figure 12-2 The Captain's Log in action

The Captain's Log allows the user to store a sequence of entries in a log file. Each entry is timestamped when it is entered and then added to the log itself, which is a long string of text. When the program is started the Add Entry page is displayed. The user can then type in some text and press "Store" to add an entry or press "View Log" to see the log entries. If they press "View Log" the program navigates to the second page, which shows the accumulated log entries. Each time a log entry is added it is annotated with the current date and time, as shown above.

We want to make the Captains Log program work in the way that the user expects. It must give the appearance of being available at all times, so that the user is not aware of way that data is being stored and loaded as required. When the user returns to the application the data that they had previously entered should be present, leaving the impression that it has always been there.

The "Captain's Log" and fast application switching

When we create an application we have to decide how it should behave when the user moves away from it. Should it store all the data in isolated storage each time, or should it only persist things if the user is closing the program? This decision has an impact on the user experience. The user might not expect information to be stored if they "jump out" of an

application using the Start button. In fact the user might think they can press Start and re-launch the Captain's Log application as a quick way of abandoning an unfinished log entry they have decided not to post.

Programmer's Point: Customers have strong opinions on application switching

Note that these are **not** technical questions. If you are making a Windows Phone program for a customer you must consult with the customer to find out what they want the program to do. The worst thing you can do is make an assumption about how it is supposed to work.

When a user presses Start to move from an application it might be best if the data is stored as a "work in progress" and not actually committed to main storage. That is how we are going to make the Captain's Log program work.

Data in the "Captain's Log" program

The Captain's Log program only contains two data elements which are both strings. They are the complete log text which is displayed on the "Log Entries" page and any half-finished log entry which is displayed on the "Add Entry" page.

```
// This is shared amongst all the pages
// It is the contents of the log itself, as a large string
public string LogText;

// This is also shared, only used by the log entry page
public string LogEntry = "";
```

The best place to put these variables is in the `App` class in the `App.xaml.cs` file. This file is the "container" for the entire application and is a good place to put data that is shared by all the pages in a program. This is the class that contains the methods that are used to manage Fast Application Switching so these methods can save and load the program data, as we shall see later.

Page Navigation in the Captain's Log Program

This application has two pages which the user will navigate between. It is important that a page always shows the correct data. We know that XAML pages are stateless (i.e. they cannot be used to record any data) and so when a page is displayed it is important to make sure that the content on the page is correct. We can use the `OnNavigatedTo` method to do this:

```
protected override void OnNavigatedTo
              (System.Windows.Navigation.NavigationEventArgs e)
{
    // When we navigate to the page -
    //put the semi-completed log text in the display

    // Get a reference to the parent application
    App thisApp = App.Current as App;

    // Put the text onto the display
    logTextBox.Text = thisApp.LogEntry;
}
```

The system will call this method for us when the user of the program navigates to the page. The code in the method gets a reference to the `App` class for this application and then loads the `LogEntry` text from this class and displays it in a `TextBox` on the screen. This makes sure that every time the page is displayed it will hold the latest application data.

We can do something similar when the user moves off the page. In this case we want to make sure that any changes that the user has made to the contents of the `TextBox` on the screen are reflected in the data our program is working with.

```
protected override void OnNavigatedFrom
                  (System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedFrom(e);

    // When we leave the page -
    // store the semi-completed log text in the application

    // Get a reference to the parent application
    App thisApp = App.Current as App;

    // Store the text in the application
    thisApp.LogEntry = logTextBox.Text;
}
```

This method is called when the user navigates away from a page. It copies the text from the textbox into the application data. In effect it is the reverse of the previous code.

Note that this technique is not just important in the context of Fast Application switching; it is also how we ensure that when the user moves between the two pages in our application the pages always display the correct information.

Saving Data when the user closes the Captain's Log

We know that when the user presses the Back button at the "top" menu in an application (in this case the "Add Entry" page) the system will close the application. When an application is closed the system will call two methods:

1. The `OnNavigatedFrom` method is called on the page the user is presently viewing.
2. The `Application_Closing` method is called to allow the application to save data.

The `OnNavigatedFrom` method copies any data from the current page into our application, the `Application_Closing` method must then save this data in isolated storage.

```
private void Application_Closing(object sender, ClosingEventArgs e)
{
        SaveToIsolatedStorage();
}
```

The application contains a method called `SaveToIsolated` storage that puts the values of `LogText` and `LogEntry` into isolated storage.

Loading Data when the user opens the Captain's Log

Now that we have code that will store data when the application closes, we have to add some code that will load the program data when it starts running. We can do this by adding to the `Application_Activated` method:

```
private void Application_Launching
                           (object sender, LaunchingEventArgs e)
{
        LoadFromIsolatedStorage();
}
```

When the application launches it now fetches its data from isolated storage. In the case of the Captain's Log this version of the method is fine.

Once this method has been completed the system will load the page that the user was at and then call the `OnNavigatedTo` method on that page. This will then take the application data and display it for the user to see.

Note that this means when a program is restarted it might not be at the MainPage.xaml page, so all your pages must be able to look after themselves and not assume that they are always arrived at as a result of navigation from another page in the application.

Handling the Start button

The code that we have written so far is how we would make an application for a Windows PC. When the application starts it loads data from a file, and when it is closed it puts the data back. However, for a Windows Phone application we have a bit more work to do. This is because a phone user might press the Start button when our program is running and cause our application to be made dormant.

When the Start button is pressed the system calls the `OnNavigatedFrom` method and then calls the `Application_Deactivated` method. We can put code in here to save data so that the user can return to the application.

```
private void Application_Deactivated
                          (object sender, DeactivatedEventArgs e)
{
    SaveToIsolatedStorage();
    SaveToStateObject();
}
```

This method seems a bit strange, in that it stores the program data in two places, isolated storage and state storage. However, it turns out that this is exactly the right thing to do. We need to store the data in isolated storage because the user may never return to the program, and they would be very upset if it lost their data. We store the data in the State object because we want to have a memory copy of the data which we can use to restore application state if the user reactivates the program.

Handling the user reactivating a program

If a user finds their way back to our program it must look as if they have never gone away from it. The job of the `Application_Activated` method is to reload data if required. We need to remember that there are two ways that an application can be activated

- It can be activated from the dormant state. In this case all the data in the program is intact and there is no need to restore anything since it is all still in memory.
- It can be activated from the tombstoned state. In this case none of the data is present and the program must reload it. It can do this by using data that it stored in the State Object when it was deactivated.

The following method does this.

```
private void Application_Activated
                            (object sender, ActivatedEventArgs e)
{
    if (!e.IsApplicationInstancePreserved)
    {
        // We are coming back from a tombstone -
        // need to restore from the state object
        LoadFromStateObject();
    }
}
```

It tests the flag that is set to true if the application is being resumed from the dormant state. If this flag is false (i.e. it is being resumed from the tombstoned state) then it loads all the program state from the State object.

Using State memory

A program can always use isolated storage to hold state information, even when it is being made dormant. However this is not always the best place to store small amounts of data that are needed to hold the state of the user interface. To make it easy and quick to hold this kind of data the Windows Phone system provides a way a program can persist small amounts of data in memory. The phone calls this "state" storage and each program has its own state storage area. This memory is kept intact even if a program is tombstoned.

The Captain's Log program can use this area to hold data when it is tombstoned rather than back to isolated storage. Then if the program is re-activated it can load the information from this area rather than using isolated storage.

```
private void Application_Deactivated
                          (object sender, DeactivatedEventArgs e)
{
    SaveToIsolatedStorage();
    SaveToStateObject();
}
```

The method above is the one that is called when a program is "tombstoned". It calls a `SaveToStateObject` method in `App.xaml.cs` to save the state of the program to memory. This method `SaveToStateObject` saves the entire state of the log by using a helper method that can save strings in the state object:

```
private void SaveToStateObject()
{
    SaveTextToStateObject("Log", LogText);
    SaveTextToStateObject("Entry", LogEntry);
}
```

The `SaveTextToStateObject` method takes two parameters, the name of the item to be stored, and the string:

```
private void SaveTextToStateObject(string filename, string text)
{
    IDictionary<string, object> stateStore =
                        PhoneApplicationService.Current.State;

    stateStore.Remove(filename);

    stateStore.Add(filename, text);
}
```

The state storage dictionary object for each application is stored in the `PhoneApplicationService.Current.State` property. To gain access to this your program must include the `Microsoft.Phone.Shell` namespace:

```
using Microsoft.Phone.Shell;
```

The state storage works like a pure dictionary of objects indexed by a string and so our program to save the data can create a local variable of this type and then just store the new item in it, making sure to remove any existing one first. Loading from the state storage works in exactly the same way too:

```
private bool LoadTextFromStateObject(string filename,
                                     out string result)
{
    IDictionary<string, object> stateStore =
                          PhoneApplicationService.Current.State;

    result = "";

    if (!stateStore.ContainsKey(filename)) return false;

    result = (string)stateStore[filename];

    return true;
}
```

This code is virtually identical to the code we wrote to load information from the isolated setting storage in the phone. The only change is that because the state storage is implemented as a pure dictionary we can use the `ContainsKey` method to see if the data was stored by the application when it was tombstoned. The fact that these methods look and function in a very similar way is actually good design.

Form Navigation and Fast Application Switching

The operating system maintains a record of the most recently used page for an application, along with the "back stack" of pages visited to reach that page. When a user returns to an application the correct page is displayed and the user will be able to perform page navigation back to previous pages in the correct way.

However, we need to remember that although page navigation is retained, the actual content of each page is lost. Fortunately it is very easy for us to make sure that a page has the correct data content by putting appropriate code into the `OnNavigatedTo` method on each page.

Fast Application Switching and Development

We can switch from an application even when it is running via Visual Studio. If the application is restarted any debugging sessions are automatically resumed in the development environment. This works even if the application is being debugged within a Windows Phone device.

You can see this at work if you press the Start button in the emulator while one of your programs is running. The Start page is opened but Visual Studio does not end debugging. If you then press the Back button you will find that the program will resume execution after being reactivated.

Forcing a Program to be removed from memory

When we are testing your program we must make sure that it works correctly even when it is removed from memory and then reactivated. We know that the Windows Phone will only remove programs when memory is short. Rather than force us to try to load up memory and force our programs to be removed, Visual Studio provides a way that we can request programs to be removed from memory as soon as they are made dormant.

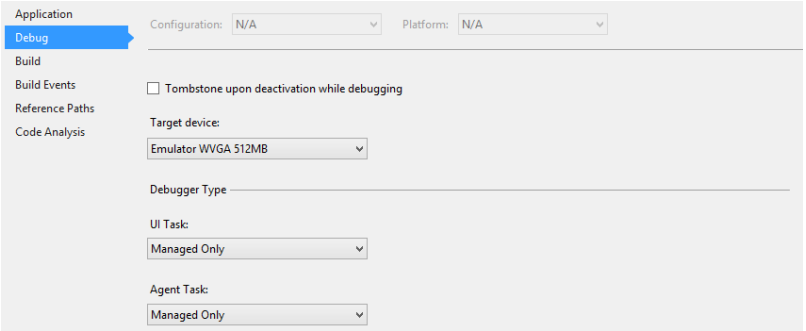


Figure 12-3 Setting for Tombstone upon deactivation

In the Debug tab of the properties for a project there is a tickbox that, if checked, requests that a program be removed from memory when it is deactivated while debugging. This allows us to test for proper behaviour of a program even when it is completely removed from memory.

Fast Application Switching and Design

It is unfortunate that we have to do so many complicated things to give the user the impression that our application never actually stops. However, it is putting in the effort to make sure that your applications work correctly in this respect as it adds a great deal to the user experience. It is worth spending some time working with this as it is a good way to brush up on your understanding of events and dictionary storage, amongst other things. It is also an interesting way to discover that the road to producing a good user experience is not always a smooth one, particularly if you are writing programs in a constrained environment.

An XNA game can also connect to events that will be generated when it is stopped or deactivated and you can make games that store game state and high scores when the user moves away from them in exactly the same way.

The solution in *Demo 01 Complete Captains Log* contains a Windows Phone logging program that works correctly when deactivated. It is worth spending some time playing with this application to see if you can "catch it out" by navigating to and from the program and seeing what data is retained.

12.2 Launchers and Choosers

Launchers and Choosers are built in behaviours that are provided to allow our programs to perform standard actions. A Launcher is means by which our program can fire off a phone feature which does not return any data. A Chooser is where a phone feature is used to select something. Your application can then do something with the item returned.

Launchers and choosers make use of deactivation in that when a program activates launcher or chooser it is automatically deactivated. When the launcher or chooser finishes the program is reactivated.

Using a Launcher

We first saw a launcher when we used one to launch the Bluetooth settings page in chapter 11. These are all the launchers, along with a brief explanation of what each does:

`PhoneCallTask`	starts the Phone application with a particular phone number and display name selected. Note that the program cannot place the call, the user must initiate this

`EmailComposeTask`	our program can set properties on an email and then launch a task to allow the user to send the message.
`SmsComposeTask`	starts the Messaging application and display a new SMS message. Note that the message is not sent automatically.
`SearchTask`	starts the Search application using a query you provide
`WebBrowserTask`	starts the Web browser and displays the URL you provide.
`MediaPlayerLauncher`	starts the Media Player application and play a given media file
`MarketplaceReviewTask`	your program can start the review process of your application
`MarketplaceDetailTask`	your program can show details of a particular application in the market place
`MarketplaceHubTask`	starts the Windows Phone Marketplace client
`MarketplaceSearchTask`	your program can search the Windows Phone Marketplace for a particular application and then show search results
`BingMapsDirectionsTask`	your program can create a set of Bing Maps directions between to places, for driving or walking
`BingMapsTask`	your program can display a map centred on the current location of the phone or a specified location
`ConnectionSettingsTask`	your program can use this task to allow users to change the connection settings of the phone
`ShareStatusTask`	your program can fill in a status message that the user can then send to social networks of their choice
`ShareLinkTask`	your program can set up a url that can then be shared by the user over the social network of their choice

Each of these tasks can be started by your application and when the task is complete your application will be re-activated. Of course there is no guarantee that our application will regain control at any point in the future. The user may instead go off and do lots of other things instead of returning to the program.

Adding an email feature to JotPad

Figure 12-4 The email button in Jotpad

We might decide to add a Mail feature to the Jotpad program. By pressing the Mail button a user can send the contents of the jotpad as an email. The image above shows how this might work.

```
private void mailButton_Click(object sender, RoutedEventArgs e)
{
    sendMail("From JotPad", jotTextBox.Text);
}
```

When the mail button is clicked the event handler takes the text out of the textbox and calls the `sendMail` method to send this. It also adds a `From` message as well. The `sendMail` method is very simple:

```
private void sendMail(string subject, string body)
{
    EmailComposeTask email = new EmailComposeTask();

    email.Body = body;
    email.Subject = subject;
    email.Show();
}
```

This creates an instance of the `EmailComposeTask` class and then sets the `Body` and `Subject` properties to the parameters that were provided. It then calls the `Show` method on the task instance. This is the point at which the JotPad application is tombstoned to make way for the email application. When the user has finished sending the email the JotPad program will resume.

To make use of the tasks a program must include the Tasks namespace:

```
using Microsoft.Phone.Tasks;
```

The solution in *Demo 02 Email JotPad* contains a Windows Phone jotter pad that can send the jotting as an email. Note that this will not work properly in the Windows Phone emulator as this does not have an email client set up. However, you can see the deactivated behaviour in action as the program tries to send an email and displays a warning instead.

Using a Chooser

These are all the choosers that are available:

`CameraCaptureTask`	starts the Camera application for the user to take a photo
`EmailAddressChooserTask`	starts the Contacts application and allows the user to select a contact's email address
`PhoneNumberChooserTask`	starts the Contacts application and allows the user to select a contact's phone number.
`PhotoChooserTask`	starts the Photo Picker application for the user to choose a photo.
`SaveEmailAddressTask`	saves the provided email address to the Contacts list Returns whether or not the save was completed.
`SavePhoneNumberTask`	saves the provided phone number to the Contacts list. Returns whether or not the save was completed
`AddressChooserTask`	your program can request that the user select an address from the contact the phone
`GameInviteTask`	your program can invite another phone user to a multiplayer gaming session
`SaveContactTask`	your program can populate a contacts entry and allow the user to save this to a contact list on the phone. Returns whether or not the save was completed.
`SaveRingtoneTask`	your program can give the user the option to save an audio file in an appropriate format as a ringtone for the phone

A chooser works in exactly the same way as a launcher with just one difference. A chooser can return a result to the application that invoked it. This is done in a manner we are very used to by now; our application binds a handler method to the completed event that the chooser provides.

Picture display

Figure 12-5 Loading and displaying a picture

We can explore how this works by creating a simple picture display application this will allow the user to choose a picture from the media library on the phone and then display this on the screen. We must create the chooser in the constructor for the `MainPage` of our application:

```
PhotoChooserTask photoChooser;

// Constructor
public MainPage()
{
    InitializeComponent();

    photoChooser = new PhotoChooserTask();

    photoChooser.Completed +=
          new EventHandler<PhotoResult>(photoChooser_Completed);
}
```

This constructor creates a new `PhotoChooser` value and binds a handler to the completed event that it generates. The completed event handler uses the result returned by the chooser and displays this in an image.

```
void photoChooser_Completed(object sender, PhotoResult e)
{
    if (e.TaskResult == TaskResult.OK)
    {
        selectedImage.Source =
                  new BitmapImage(new Uri(e.OriginalFileName));
    }
}
```

This method sets the source of the image to a new `BitMapImage` that is created using the address of the photo result. Note that the `TaskResult` property of the result allows our program to determine whether or not the user chose something to return.

The final link we need to provide is the event handler for the Load button that starts the process off.

```
private void loadButton_Click(object sender, RoutedEventArgs e)
{
    photoChooser.Show();
}
```

This method just calls the `Show` method on the chooser to start the choosing process running.

The solution in *Demo 03 PictureDisplay* contains a Windows Phone program that you can use to select an image from the media storage. This program will work on the Windows Phone emulator but it does not have any images in is media store. It can be used to select images on a real device however.

What We Have Learned

1. The Windows Phone operating system provides a single tasking model of execution for applications. An application may be deactivated at any time and replaced by another. The Start and Back keys on the phone device allow a user to deactivate one running application and start another by pressing Start, and then return to the deactivated application by pressing Back. The user can also hold down the Back button to display a list of deactivated applications from which to choose.
2. A deactivated application can either be in memory (dormant) or not (tombstoned). When a dormant application is reactivated it has all memory intact. A tombstoned application must reload all data when it is reactivated.
3. There are four events which are produced in an application as it runs on the phone. They are “Launched” (fired when the application starts), “Closed” (fired when the application is closed), “Deactivated” (fired when the application is deactivated) and “Activated” (fired when the application is activated from a dormant or tombstoned state).
4. Windows Phone provides in memory storage mechanism which can be used to store live data when an application is deactivated. This can be read back if the application is ever reactivated.
5. Programs can use Launchers and Choosers to interact with phone subsystems. A Launcher starts an external application (for example and email send) and a chooser launches an external application that returns a result (for example selecting a picture from the media store). A program may be deactivated while the external application runs.

13 Background Agents and Live Tiles

In this chapter we are going to explore how you can create program elements that run in the "background" on Windows Phone. We are also going to discover how background programs can attract the attention of the user by updating Live Tiles on the display.

13.1 Background Processing

We know that a Windows Phone will only allow one program at a time to be active on the phone. However, there are occasions when it would be very useful for some part of an application to be active when it is not visible to the user. The Windows Phone operating system provides this in the form of "background processing" features, where a developer can create code which will run when their application is not active.

Note that this is **not** the same as allowing a given program to run in the background. The background components are entirely separate, have no user interface and have strict limitations placed on what they can and can't do. They only run in specific situations so that the phone operating system can ensure that they will have an impact on the user experience. The phone operating system will shut down background tasks if memory, battery or network connectivity is compromised. The user can also disable background tasks for a given application.

We should therefore regard a background task as a piece of "icing" on an application. It should not be a fundamental part of how the application works, since it may never get to run and the user can disable it. Examples might be sales applications that display a message when a requested item appears in stock, or news applications that display the latest headlines on a tile on the Start screen.

Background and Periodic Scheduled Tasks

There are a number of different types of background process; we are going to start by considering the two general purpose background tasks, the *periodic* and the *resource intensive* tasks. We create these tasks in the same way, however the situations in which the code in them gets to run is slightly different.

- A periodic task is allowed to run for a short amount of time (up to 15 seconds) at regular intervals - usually every half an hour or so
- A resource intensive task is allowed to run for longer times (up to 10 minutes) when the phone is locked (i.e. the user is not doing anything with it), connected to a power source and has high quality network connectivity available

The periodic option is useful for updating things like weather displays, status information or location tracking. The resource intensive option is useful for downloading/uploading updates to databases, performing data processing or compression.

Agents and Tasks

When talking about these kinds of agents we need to sort out terminology. A *Task* is the container that is managed by the operating system and is run when circumstances allow. An *Agent* is the actual code payload that does the work. When we create an agent we make a single item which will run in either periodic or resource intensive mode. When the agent gets control it can determine which mode it is running in.

The Captain's Log Location Tracker

To investigate how to use an agent in an application we could return to the "Captain's Log" program.

Figure 13-1 Captains Log Location Tracker

The program can be improved by adding a location tracking feature. When this is enabled it will save the location of the phone using a periodic task to regularly read position information from the GPS device in the phone, add a timestamp and save it in the log file. The idea is that the background task will save the phone position information when the Captain's Log program is not active.

Adding a Scheduled Task to Captains Log

A scheduled task is added as an extra project to a Windows Phone solution.

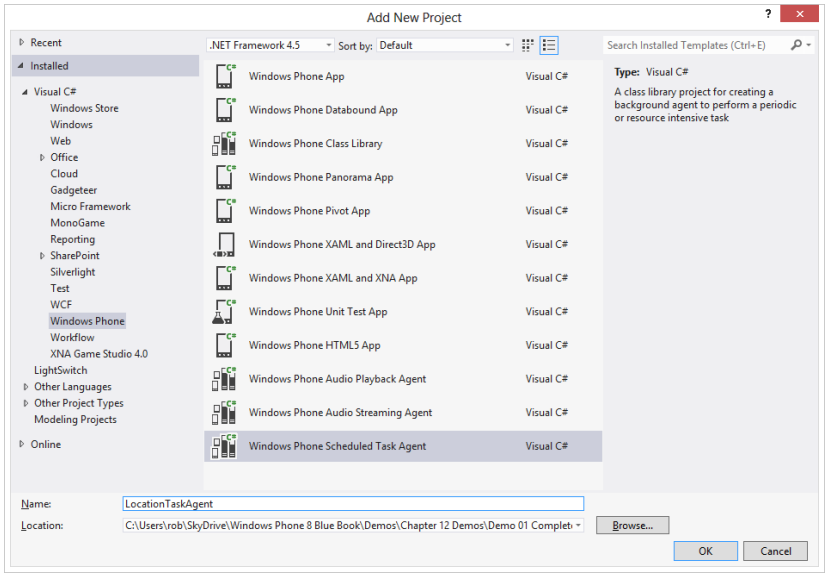


Figure 13-2 Adding a scheduled task agent

Note that it is added as a Scheduled Task item, which could be either periodic, resource intensive, or both. Once it has been added the background task will show up as another project in the solution for the application.

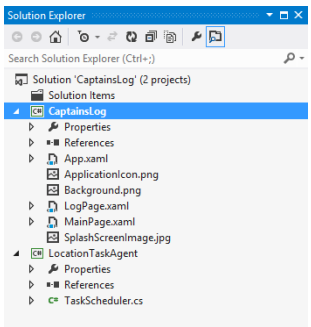


Figure 13-3 The LocationTaskAgent project

A given application can only have one Scheduled Task associated with it. When the application is built the project that contains the code for the scheduled task is added to the other parts of the application.

The main Captain's Log program will create a scheduled task using the code in the agent project. For this to work we need to make the main application able to use the agent code. In Visual Studio this means adding the output of the `LocationTaskAgent` project to the references used by the foreground project. We have already seen how to do this when we considered solutions made up of multiple projects, this is another example of the technique in action.

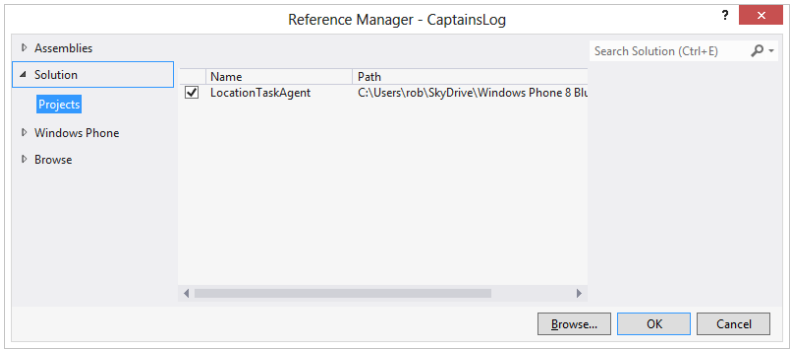


Figure 13-4 The Reference Manager

To open the Reference Manager in a project find the References folder in Solution Explorer and right click it.

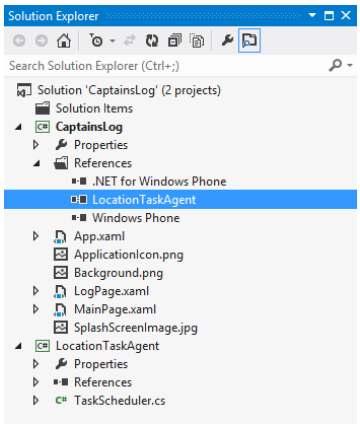


Figure 13-5 Added reference

Figure 13-5 shows the reference after it has been added to the project.

Background Task Agent Code

When we create a background task Visual Studio makes an empty template that we can fill with agent code.

```
namespace LocationTaskAgent
{
    public class ScheduledAgent : ScheduledTaskAgent
    {
        protected override void OnInvoke(ScheduledTask task)
        {
            //TODO: Add code to perform your task in background
            NotifyComplete();
        }
    }
}
```

We write the code that performs the task in the `OnInvoke` method. When the method finishes it must call the `NotifyComplete` method to stop this run of the task. Note that we don't ever control when, or even if, this code gets to run. When we register the Agent with the phone we will request either periodic, resource intensive, or both execution scenarios. Then, if the user has allowed background tasks and the time is right, our program gets to run.

Our agent code can determine what kind of task is running by checking the type of the `ScheduledTask` that was passed to `OnInvoke`:

```
if (task is PeriodicTask)
{
    // Is running as a periodic task
}
```

Loading Application Data in a Background Task Agent

In the case of the Captain's Log, the task must load the log string from isolated storage, add location information to it and then store the log string back again. The application and its background task both share the same isolated storage area. Note that there is no question of the two fighting over access to this resource, as a background task will never run if the application is already active.

```
protected override void OnInvoke(ScheduledTask task)
{
    string message ="";

    string logString = "";

    if (loadTextFromIsolatedStorage("Log", out logString))
    {
        message = "Loaded ";
    }
    else
    {
        message = "Initialised ";
    }

    //When we get here we have a log string - add location to it

    NotifyComplete();
}
```

This part of the background task agent will call a method to load the log text from isolated storage. This is the same code that is used to store and load the data when the foreground program runs. If the log is initially empty (i.e. the agent is running before the application has stored anything in isolated storage) then the string in the message variable is set appropriately.

Accessing Location Information in a Background Task Agent

Now that we have the log string, the next thing to do is find the position of the phone and add this to the log text.

```
protected override void OnInvoke(ScheduledTask task)
{

    //When we get here we have a log string - add location to it

    GeoCoordinateWatcher watcher = new GeoCoordinateWatcher();
    GeoPosition<GeoCoordinate> position = null;

    watcher.StatusChanged +=
      new EventHandler<GeoPositionStatusChangedEventArgs>
      (delegate(object sender, GeoPositionStatusChangedEventArgs e)
        {
            if (e.Status == GeoPositionStatus.Ready) {
                position = watcher.Position;
                GPSDoneFlag.Set();
            }
        });

    GPSDoneFlag.Reset();

    watcher.Start();

    // Wait for 10 seconds to give time to get position
    GPSDoneFlag.WaitOne(10000);

    // When we get here we the position or we timed out

}
```

This part of the background agent creates a new `GeoCoordinateWatcher` and starts it running. The `GeoCoordinateWatcher` exposes a `StatusChanged` message that we can connect to. This event is fired whenever the GPS reader changes state. The code above binds a handler to this method that looks for a change to the `Ready` state. When the GPS reaches this state it reads the position value and then sets a flag to allow a waiting thread to continue. We used the same technique when our programs were waiting for TCP network requests to complete in Chapter 7. It allows us to write programs that can wait for the completion of a system action. If the GPS position is available the value is copied into the `position` variable.

Note that for a background task agent the behaviour of the GPS is slightly different. Since it would be very inefficient to start the GPS system for every background process the system instead makes a copy of the location information at regular intervals and uses this for any agents that request position information. This means that rather than having to wait for an event to indicate that position information is available (which is usually how a GPS watcher would be used) this program can read the position information immediately to create the log string.

Note that the code above works fine on the Windows Phone device but you might notice problems using the emulator because the emulator position information is not passed into background tasks.

Building a Log Message

Now that we have the position information the next thing we have to do is create a log message and store it back in isolated storage. At the moment the data is stored as a simple pair of values, it would be easy to save it as XML instead.

```
protected override void OnInvoke(ScheduledTask task)
{
    // When we get here we got the position or we timed out
    string positionString ;

    if (position != null)
        // If we get here we got the position or we timed out
        positionString = position.Location.ToString();
    else
        positionString = "Not Known";

    DateTime timeStamp = DateTime.Now;
    string timeStampString = timeStamp.ToShortDateString() +
                             " " + timeStamp.ToShortTimeString() +
                             System.Environment.NewLine;

    logString = logString + timeStampString + positionString;

    saveTextToIsolatedStorage("Log", logString);

    // When we get here we can display a toast message
}
```

The above code builds a text message which contains the position value or “Not Known” if this was not available. It then adds this to the end of the log string and saves this back to isolated storage.

Showing a “Toast” Message from a Background Agent

Once the location has been stored the location logger could tell the use that this has been performed. A background task is not able to communicate directly with the user, but it can display popup messages or change the appearance of tiles on the Start screen.

Figure 13-6 Pop up location display

The screenshot in Figure 13-6 shows the location display. Note that because it is running in the emulator the location data was not available to the background agent. The toast message shows a simple popup that was produced by the above code. The phone will play an audible warning and the user can start our application by tapping the popup. The code to display a toast popup is very simple. Once the instance has been created we just have to set the title and the message to be displayed and then call the Show method.

```
protected override void OnInvoke(ScheduledTask task)
{
    // When we get here we can display a toast message

    ShellToast toast = new ShellToast();
    toast.Title = "Captain's Log";
    toast.Content = message + positionString;
    toast.Show();

    // Now we need to schedule the task for debugging
}
```

We can modify the toast to give it a uri which identifies a particular page in our application which should be navigated to if the user selects it. This would make it possible for an alert from a background task to take the user directly to the part of the program concerned with that alert.

Debugging with a Background Agent

You might thing that debugging a background agent would be difficult, but actually this is not the case. Visual Studio will allow us to use breakpoints and single step through background agent code in exactly the same way as we would for any other part of our program.

When we are debugging we don't want to have to wait until the conditions are right for a background task to run. For this reason the program provides a method that we can call to launch a background task for testing. We can use conditional compilation to trigger a call of a method that will run a background agent when we are debugging.

```
#define DEBUG_AGENT
```

By defining this symbol we can cause the compiler to process code that will schedule the agent to run.

```
protected override void OnInvoke(ScheduledTask task)
{
    // Now we need to schedule the task for debugging

#if DEBUG_AGENT
    ScheduledActionService.LaunchForTest(task.Name,
                                         TimeSpan.FromSeconds(60));
#endif

    NotifyComplete();
}
```

The code above would make the background agent run again in 60 seconds time. Note that this does not request repeated execution of the agent, although because each time the agent runs it requests another run in 60 seconds, it has that effect.

The final call of `NotifyComplete` is how the background task agent informs the Windows Phone runtime system that it has completed processing.

Managing the Background Agent Task

Now we have created the code for our background agent we can now consider how we are going to get it running. The Windows Phone operating system contains a class called `ScheduledActionService` that manages all scheduled tasks. We can call methods on this class to search for active tasks and start and stop them. An active task has a particular name that we look for. Of course we can only control the tasks that our application runs; we are not allowed to see tasks started by other applications.

```
PeriodicTask periodicTask = null;
string periodicTaskName = "CaptainTracker";

private void StartTracking()
{
    periodicTask = ScheduledActionService.Find(periodicTaskName)
                   as PeriodicTask;

    // If the task already exists and the IsEnabled property
    // is false, background agents have been disabled by the user
    if (periodicTask != null && !periodicTask.IsEnabled) {
        MessageBox.Show(
                      "Background agents disabled by the user.");
        return;
    }

    // If the task already exists and background agents are enabled
    // for the application, we must remove the task and then add it
    // again to update the schedule
    if (periodicTask != null && periodicTask.IsEnabled) {
        RemoveAgent(periodicTaskName);
    }

    periodicTask = new PeriodicTask(periodicTaskName);

    // The description is required for periodic agents. This is the
    // string that the user will see in the background services
    // Settings page on the device.
    periodicTask.Description = "Log Tracker";
    ScheduledActionService.Add(periodicTask);

    // If debugging is enabled, use LaunchForTest to launch the
    // agent in one minute.
#if(DEBUG_AGENT)
    ScheduledActionService.LaunchForTest(periodicTaskName,
                                         TimeSpan.FromSeconds(60));
#endif
}
```

The above code will start an agent as a periodic task and then cause it to run in 60 seconds if we are debugging. Note that the task must have a particular name and that the user can disable background agents for a particular application if they wish. We could start a resource intensive task by creating a `ResourceIntensiveTask` instance instead of a `PeriodicTask`. If we want to create a task that runs in both situations we can create a `ScheduledTask`.

The solution in *Demo 01 Location Logger* contains a Windows Phone program that performs location logging using a background task. When the program starts running it checks for an existing tracking agent and allows the user to stop or start the tracking. It uses the "LaunchForTest" debugging mode to update the location every minute.

13.2 Adding a Live Tile to an Application

The problem with a Toast message is that if the user is not around to see it, the message will not be displayed. However, there is a way that an application can display a more persistent message to the user. In order to use this feature the user must "pin" the application to the Windows Phone start screen. Then the application can update the "Live Tile" that is displayed when the application is pinned.

There are three sizes of Live Tile and an application can use any, or all, of them to deliver update information to the user. The Live Tile can contain text, an image and an "update count" which the user can view. A tile can also have a "flip" side, the tiles themselves flip every six seconds or so when they are displayed. One type of tile can also cycle between up to six tile images. A Live Tile can be constructed which starts an application or which contains a "deep link" to a page within the application.

Creating a Flip Tile

A flip tile has a picture and a counter on the front and, for the normal and wide tiles, a back which can contain another picture, a heading and a small amount (up to 80 characters) of text. The images that are displayed are loaded form a Uri and so can be local resources on the phone or image obtained from the internet.

The small and medium tiles are square, but the large tile is rectangular.

The Small Tile

The small tile is nominally 159 pixels square, but the system will scale the image if required.

Figure 13-7 A small tile in amongst other small tiles

Figure 13-7shows a small tile on the phone display, within four other tiles. The tile can contain a counter value, in this case 5. This number lets the user know that some items (email messages, orders, contact requests) are waiting for them in the application. If the counter value is 0 it is not displayed. Note that there is no text on the small tile.

The Normal Tile

The normal tile is nominally 336 pixels square, but again a larger image will be scaled for display. You can use the same image for both tiles if you wish, that is what I have done.

Figure 13-8 A "normal" tile, front and back

Figure 13-8 shows a normal tile. This has a title and a flower image on the front and a gradient fill, title and body text on the back. The front of the tile also has the counter value.

The Wide Tile

The wide tile is nominally 691 pixels wide and 336 pixels high.

Figure 13-9 A Wide tile front and back

Figure 13-9 shows the front and back of a wide tile. The back of the wide tile can have a separate title and extended text content as shown.

There are some useful design assets available that can help you with the construction of your tiles and their content. It is important to remember that the users will have your tile in front of them for a lot of time and that it should look good.

Display a Flip Tile

To create and display a flip tile a program just needs to create the `FlipTileData` object and then update the tile with the new information.

```
ShellTile primaryTile = ShellTile.ActiveTiles.FirstOrDefault();

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    FlipTileData newTile = new FlipTileData()
    {
        Title = "Demo Tile",
        Count = 5,
        BackgroundImage = new Uri("/Assets/Tiles/FlowersSQ.png", UriKind.Relative),
        SmallBackgroundImage = new Uri("/Assets/Tiles/FlowersSQ.png",
UriKind.Relative),
        WideBackgroundImage = new Uri("/Assets/Tiles/FlowersWide.png",
UriKind.Relative),
        BackTitle = "The Back",
        BackContent = "Hello from the back",
        WideBackContent = "Hello from the back in wide screen",
        BackBackgroundImage = new Uri("/Assets/Tiles/NormalBack.png",
UriKind.Relative),
        WideBackBackgroundImage = new Uri("/Assets/Tiles/WideBack.png",
UriKind.Relative)
    };

    primaryTile.Update(newTile);
    base.OnNavigatedTo(e);
}
```

This is the code that creates the tiles that were shown above. You can map all the elements to the items on the tiles.

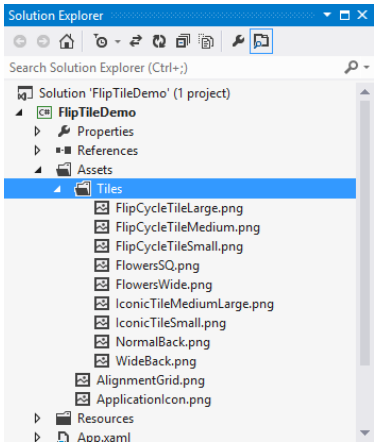


Figure 13-10 The tile assets in Solution Explorer

Figure 13-10 shows where the assets are stored for the demo tile application. Note that there are empty placeholder images that you can fill in with data. These can also be used to give you an idea of size.

You can also create `CycleTile` items that can cycle through up to 8 images. The images are supplied as a list of Uri values and the phone display will cycle through them.

Using Tiles

An application can have a Primary tile and a number of Secondary tiles that link to subpages within the program. It is possible for a program to use the `WriteableBitmap` class to create a tile image which is subsequently displayed for the user. This provides a great deal of flexibility.

You can update live tiles from within background tasks so that users can be alerted to new blog posts, news events, items arriving in stock or anything that your application might want to tell them about.

The solution in *Demo 02 FlipTileDemo* contains a Windows Phone program that displays the tiles shown above.

13.3 File Transfer Tasks

It is also possible for us to create a background task to transfer files to and from our application's isolated storage area. The transfers will take place when the application is not running and when an application is restarted it can monitor the state of any active downloads and display their status. Files can be fetched from HTTP or HTTPs hosts but at the moment FTP is not supported by this system.

Background File Transfer Policies

There are a number of policies that govern the amount of data that can be transferred by an application using the background file transfer agent:

- Maximum Upload file size: 5Mb
- Maximum Download file size over cellular (mobile phone) data: 20Mb
- Maximum Download file size over Wi-Fi: 100Mb

A given transfer can modify these values using `TransferPreferences` if different settings are required for it.

Creating a Background Transfer

The background transfer services are all defined in the background transfer namespace:

```
using Microsoft.Phone.BackgroundTransfer;
```

We start a transfer by creating a `BackgroundTransferRequest` object and then adding it to the ones presently managed by the `BackgroundTransferService`. The transfer is given a name so that next time the application runs it can locate the transfer and find out the progress of the transfer.

It is also possible to bind to events that the transfer generates, so that a running application can display progress information.

Note that it is up to our application to manage the active transfers and remove ones that have completed.

Figure 13-11 Picture Fetch Demo

The solution in *Demo 03 File Transfer* contains a Windows Phone program that will fetch an image file from the internet and display it. The program also displays status information as the file is downloaded.

13.4 Scheduled Notifications

While it is not possible for an application to create calendar appointments it can create scheduled notifications. These are retained when the phone is switched off and can cause a toast popup or a change to a Live Tile. Note that a scheduled notification can't start your application running, but if the user taps on the notification your application will be started. The notifications can be made to fire once, or repeat at given intervals.

Reminders are created in a similar way to File Transfers, and can be located and managed by our applications by names that we give them.

Figure 13-12 Using the Egg Timer

The solution in *Demo 06 EggTimer* contains a Windows Phone program that will time from 1 to 5 minutes.

Audio Playback Agent

An application can also create an audio playback agent. This integrates with the normal playback controls and allows an application to play music when it is not running. A background agent project must be created and added to a Windows Phone solution in the same way as a background task.

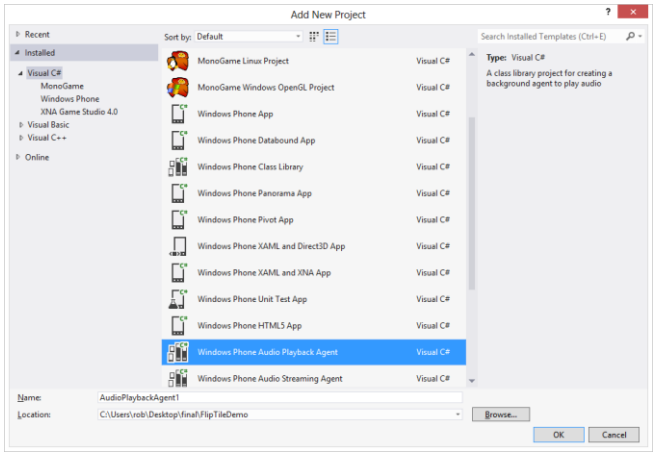


Figure 13-13 Adding an Audio Playback agent

The playback agent can play music from the isolated storage for the application but it does not have access to the media library on the phone. However it can be used to playback from a network media source.

What We Have Learned

1. An application can create background agents that can run when the application is not active. These can be *periodic* (run for a short time every 30 minutes) or *resource intensive* (run when the phone is not in use, externally powered and has a good network connection).
2. The user of the phone can determine which, if any, of the background agents allocated to applications are allowed to run. If the user chooses, or the phone enters a power saving mode, the background agent for an application may be stopped. This means that a background agent should never provide a core behaviour for your application, it is best regarded as a piece of "icing on the cake".
3. The agent is a separate project that is added to the application solution. The code in the application can determine whether it is running in periodic or resource intensive mode.
4. Background agents can display popup "toast" notifications to users.
5. Background agent and foreground applications may run simultaneously, and so they should make use of synchronisation to ensure that there is no conflict over system resources.
6. Background agents can use "Live Tiles" on the Windows Phone start screen to display status information to the user. To view these the user must have "pinned" the application to the start screen.
7. An application can create background file transfers that will run when the application is not active.
8. An application can create scheduled notifications which will run at particular intervals. These will appear even when the application is not running.

14 Marketing Windows Phone Applications

We now know enough to make applications that run on the Windows Phone device. In this section we will to take a look at what takes a program and turns it into a "proper" application. This includes a variety of topics, from how to give your program a custom splash screen and icons to how an application can create and manage background tasks.

14.1 The Windows Phone Icons and Splash Screens

At the moment all the programs that we have written have had the same icon and "splash screen". These have been the "empty" ones that Visual Studio created when it made a new project. There are two images that Windows Phone uses to identify your program when it is stored on the phone. To understand how these are used we have to learn a little about how Windows Phone users find and run programs on the device.

A Windows Phone has a "Start" screen which is displayed whenever the user presses the "Start" button on the device:

Figure 14-1 Windows Phone start screen

The user can then touch any of the tiles on the start screen to run the program behind that tile. Alternatively they can slide the start screen to the left to move to a list of applications that are installed on the device.

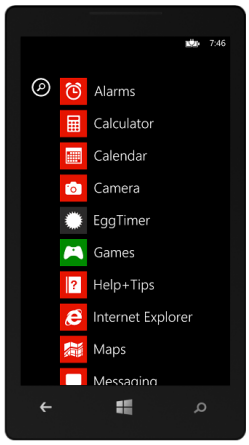


Figure 14-2 Windows Phone programs screen

Then the user can scroll up and down the application list and select the program that they want to run. If the user wants to be able to start one of their applications from the Start screen they can “pin” it to the start screen.

Program icons

If you create a new application Visual Studio will create a project that contains “default” versions of these two icons:

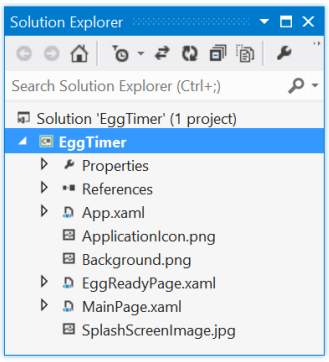


Figure 14-3 Icon files in the Visual Studio Solution

The file `ApplicationIcon.png` holds the image to be used in the application list. The file `Background.png` holds the image to be used when the application has been “pinned” to the Start menu.

Customising Icons

If you want to customise the program icons (and you should) you can open and edit these files with any paint program that you like. Remember that they are stored in the project folder along with all the project files. However, you must make sure that you retain the file type and the image dimensions of the originals.

From my experience it turns out that designing icons like this (particularly small ones) is something of an art form. For this reason if you can find an artist to do the design for you this will definitely be a good idea. If you have to do the work yourself I would advise you to make the design as simple as possible and make sure that it gives the user a good idea of what your program does. Remember that the icon for your program is also used to promote it in the store, so the better and slicker the icon looks the more chance there is of people downloading and buying your program.

If you use the Marketplace test kit to prepare your application for sale you can load all your artwork and icons into this. It will also check the size and format of each image. The images are all held in the PNG format which supports transparency. Any transparent elements in your icon images will show through the currently selected background. This means that you need to be a bit careful about the design of your image. You also need to make sure that the icon that you use will work in both the “light” and “dark” colour schemes that Windows Phone supports.

Figure 14-4 Light and Dark themes on Windows Phone

Figure 14-4 shows the two themes in action. If you submit an application for sale in the Windows Phone Marketplace it will be checked to make sure that any icons that you create are useable in light and dark.

Splash Screens

A "splash screen" is something that a program "splashes" onto the display as it starts up. These can be used to "brand" your application by showing your company logo when your program starts. Splash screens are also useful to improve the experience of program users when your program starts running. At the very least a splash screen will give them something nice to look at as your program loads.

Default Splash Screens

The run time system has a splash screen behaviour built in. A new project actually contains an image file which is displayed during the interval between a program being started and `MainPage` being displayed on the screen. If you have run programs on the emulator you may have seen this for a brief instant at the very start of a program run.

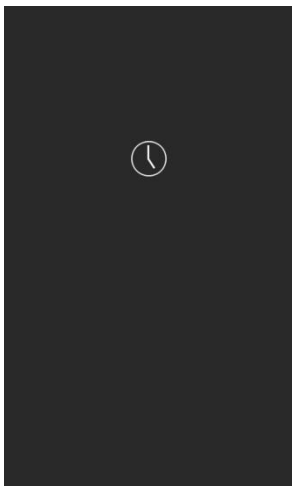


Figure 14-5 Default splash screen

If your programs become larger, with more elements on the page and more resources to be loaded, this page may be displayed for more time. You can make your own version by modifying the `SplashScreenImage.jpg` file in your project.

14.2 Preparing an Application for Sale

In this section we are going to focus on the things that we can do to prepare an application for sale.

Application Analysis

Modern computers are so powerful that they are often able to compensate for badly written software. It is frequently cost effective to buy a faster computer rather than optimise a program. However, when developing for Windows Phone we do have to worry about performance a lot more than we would for a desktop application. This is because the processor power is limited and the device itself has less memory than a PC.

The Windows Phone SDK provides a set of performance analysis tools that we can use to find out what our program is doing and how will it performs. They can give an idea of the loading our programs place on the processor, the screen update rate (how rapidly our program is updating the screen) and memory usage. We can even find out very low level detail about which particular methods are being called in our program. This is very useful as it means we can focus our optimisation on those parts of the program that are consuming most of the processor time.

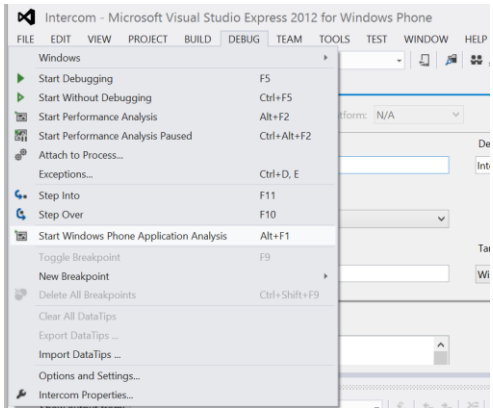


Figure 14-6 Starting the Application Analysis tool

The analysis tools are started from the Debug menu in Visual Studio. When they are opened they display a configuration screen:

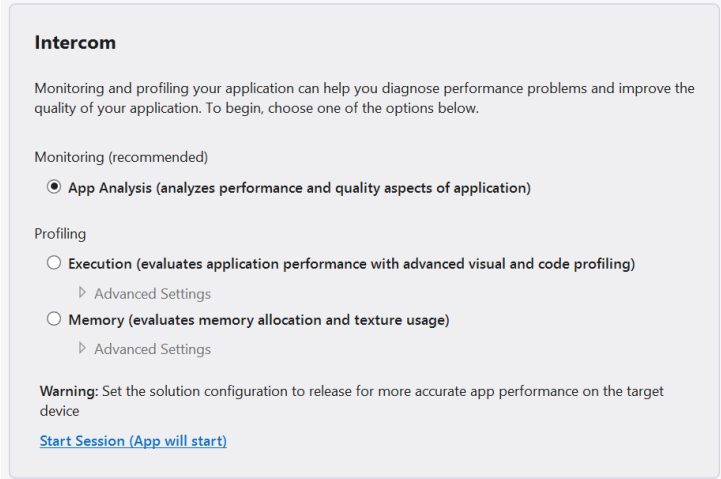


Figure 14-7 Starting the Application Analysis

There are a number of different options we can select, depending on whether we are interested in performance or the memory usage. When we have selected our options the program will be run within the performance analysis too. The program can run inside the emulator or a hardware device. In either case the system produces a log file of the session which is stored as part of the Visual Studio Project. This makes it easy to look back at previous performance tests and determine the effect of changes that we make.

When the program is stopped at the end of a performance analysis session Visual Studio will write the log file and then allow us to explore the results.

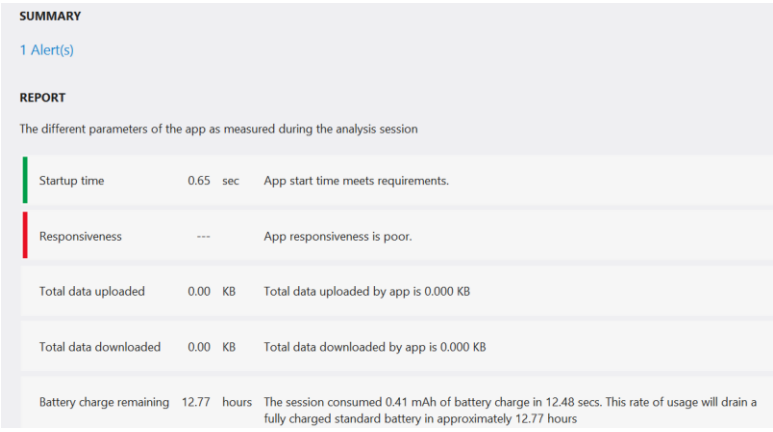


Figure 14-8Application Comments

This is the report for the Bluetooth connect application. It looks like I need to work on performance….. We can see that for the first few seconds the program is initialising the camera device. Then it starts to process the video stream and the CPU usage increases. The program also shows the frame rate that the application is achieving. The program itself does not use much memory, although it allocates some video buffers once the camera has initialised.

This is a very high level view of the information, it is possible to drill down and take a much more detailed view. Things to watch for are sudden jumps in CPU usage, a steady increase in memory use or a large number of Garbage Collection (GC) events. These might indicate that an application is not being very sensible with the way that it uses the phone resources.

Creating a XAP File for Application Distribution

We have already seen that an application is described by a Visual Studio solution. In Chapter 3 “Running Windows Phone Applications” we saw that an entire application is stored in a single XAP file. This is the file that is transferred into a Windows Phone device when the program is deployed. It is also the file that is submitted to the Windows Phone Marketplace when we want to sell our application.

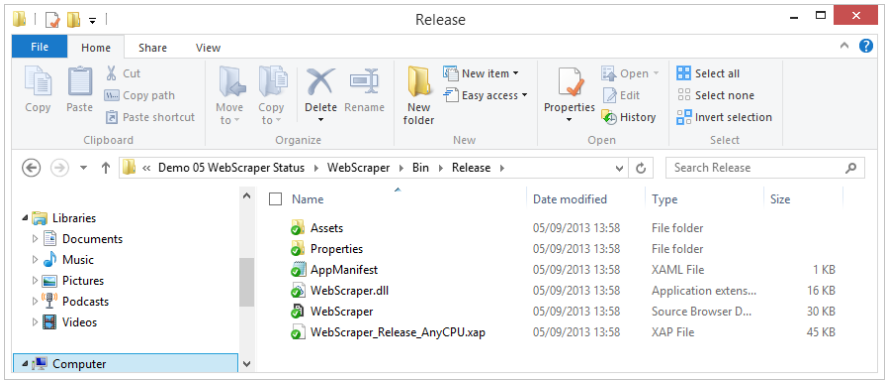


Figure 14-9 The XAP file in the Release Binary Folder

This file contains all the program assemblies and resources along with manifest files that describe the application. The `WMAppManifest.xml` file lists the capabilities of the phone that the application

```
<?xml version="1.0" encoding="utf-8"?>
<Deployment xmlns="http://schemas.microsoft.com/windowsphone/2012/deployment"
AppPlatformVersion="8.0">
  <DefaultLanguage xmlns="" code="en-US" />
  <App xmlns="" ProductID="{c91a4e46-0b98-45a1-a2dc-f6c7e491c491}"
Title="Procrastinator" RuntimeType="Silverlight" Version="1.0.0.0" Genre="apps.normal"
Author="ProcrastinationChallenge author" Description="Measure your procrastination
skills." Publisher="ProcrastinationChallenge" PublisherID="{1676fdde-0fb0-41af-985f-
2dad0fa39b0d}">
    <IconPath IsRelative="true" IsResource="false">i202.png</IconPath>
    <Capabilities>
      <Capability Name="ID_CAP_WEBBROWSERCOMPONENT" />
      <Capability Name="ID_CAP_NETWORKING" />
    </Capabilities>
    <Tasks>
      <DefaultTask Name="_default" NavigationPage="MainPage.xaml" />
    </Tasks>
    <Tokens>
      <PrimaryToken TokenID="ProcrastinationChallengeToken" TaskName="_default">
        <TemplateFlip>
          <SmallImageURI IsResource="false" IsRelative="true">i300.png</SmallImageURI>
          <Count>0</Count>
          <BackgroundImageURI IsResource="false"
IsRelative="true">i300.png</BackgroundImageURI>
          <Title>Procrastinator</Title>
          <BackContent></BackContent>
          <BackBackgroundImageURI></BackBackgroundImageURI>
          <BackTitle></BackTitle>
          <DeviceLockImageURI></DeviceLockImageURI>
          <HasLarge>false</HasLarge>
        </TemplateFlip>
      </PrimaryToken>
    </Tokens>
    <ScreenResolutions>
      <ScreenResolution Name="ID_RESOLUTION_WVGA" />
      <ScreenResolution Name="ID_RESOLUTION_WXGA" />
      <ScreenResolution Name="ID_RESOLUTION_HD720P" />
    </ScreenResolutions>
  </App>
</Deployment>
```

Above you can see the default `WMAppManifest.xml` file that was created for my “Procrastination Challenge” application. I’m still thinking about releasing it. This version of the file indicates that the application will use two of the phone capabilities although these are actually set by default as the program doesn’t need them.

When an application is submitted to the Marketplace the content of this file is checked against the calls that the application makes. If the application uses resources that are not specified in this file it will fail validation.

When a customer installs a program they are told the features of the phone that it will use and are asked to confirm this. The idea is to stop programs with unexpected behaviours being used by unsuspecting customers, for example a picture display program that also tracks the position of the user.

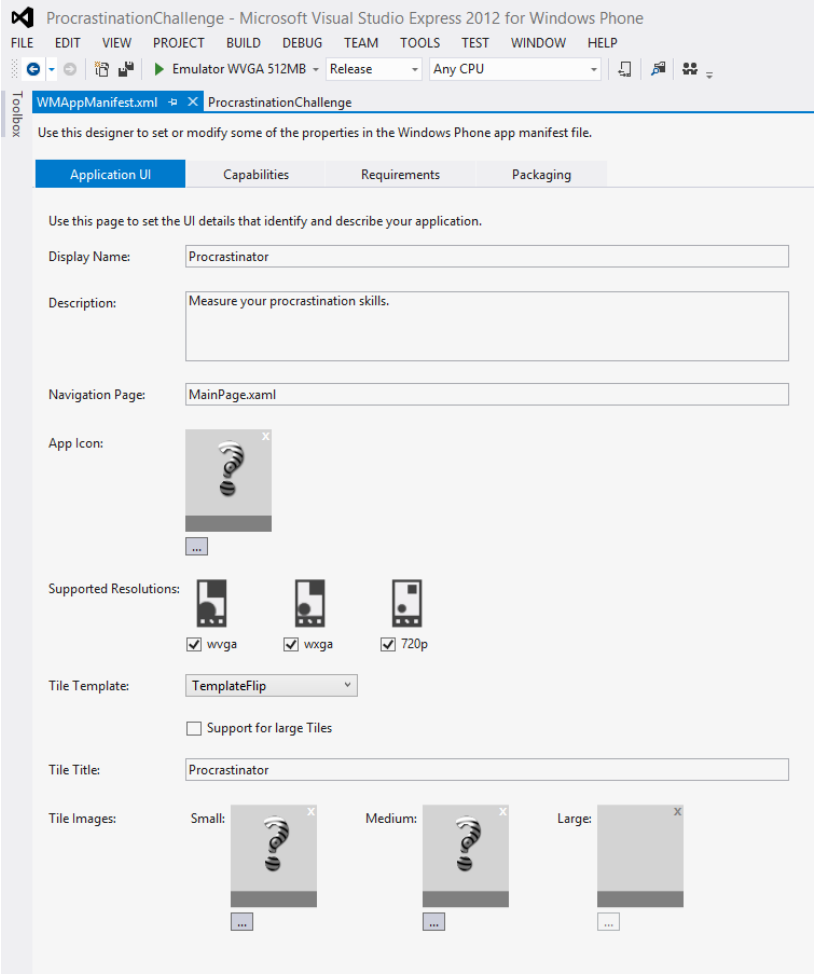


Figure 14-10 Editing the App Manifest

We can edit the manifest file from within Visual Studio. Figure 14-10 above shows one of the four editing tabs for the manifest file. This is the one where you create the description of the application, set the icons for the application and identify which screen resolutions that the application can use. You must also provide icons for each of the tile sizes. Note that for this program I have not enabled support for Large tiles.

Note that when we produce a version of the program for upload to the marketplace we need to make sure that it has been compiled in Release mode, not Debug.

Creating Application Tiles and Artwork

Before we can submit an application we also need to prepare the tiles that will denote the application on the device and we also have to prepare some artwork for display on the Windows Phone Marketplace. The icons must have particular sizes and be stored in Portable Network Graphics (PNG) files. The actual sizes that you need are:

- A small mobile app tile icon (required) which is used in the phone Windows Phone Marketplace, 99 x 99 pixels in size.
- A large mobile app tile icon (optional) which is used in the phone Windows Phone Marketplace, 173 x 173 pixels in size.
- A large PC app tile icon (required) which is used in the phone Windows Phone Marketplace, 200 x 200 pixels in size.
- Background art (optional) which is used in the Background panorama in the Marketplace entry for your application, 1000 x 800 pixels in size.

You must also create at least one screenshot which is 800x480 pixels in size. You can actually upload several screenshots, this is a very good idea as it gives potential purchasers a better idea of what your program is like.

It is a very good idea to get the help of a graphic designer to prepare the icons and background art for your program.

Registering a Phone as a developer device

You will want to do your final testing of your program on a real device. This is not strictly necessary, in that it is perfectly possible to successfully build and sell a program without running it on actual hardware. However, it is very useful (and impressive) to be able to see your code running on real hardware. This is particularly true when creating games, when you need to be able to evaluate the gameplay.

A Windows Phone can normally only run programs that have been obtained via the Marketplace. However a marketplace member can register a phone as a developer device so that it can be used to run programs that are loaded into it from Visual Studio.

Fee paying Marketplace developers can register up to three devices and deploy up to ten of their own programs on each device. Student members who joined via DreamSpark can register one device. Anyone with a Microsoft account can register a phone for development and deploy two applications on that device.

The registration is performed using an application that is installed as part of the Windows Phone software development kit. This application can register and de-register phones. To find the application on your Windows 8 machine press the Start button and search for "Developer". Then select the Developer Unlock program from the ones that appear.

Before you run the program, make sure that your phone is plugged into your Windows 8 PC and the Windows Phone application (the one you can use to transfer media to and from your phone) is not running. You also need to open the lock screen so that the Windows Phone is showing the start page.

The PC you are using to unlock the phone must also have a working network connection, as the program will go online to update your Microsoft account information when the phone is registered.

Figure 14-11 Registering a phone for development

Figure 14-11 shows the display from the unlock program. The help from this program is no longer completely correct, as you do **no longer** need a Windows Phone Marketplace Account to register a phone for development. What you do need is a Microsoft Account. To register the phone just click the Register button.

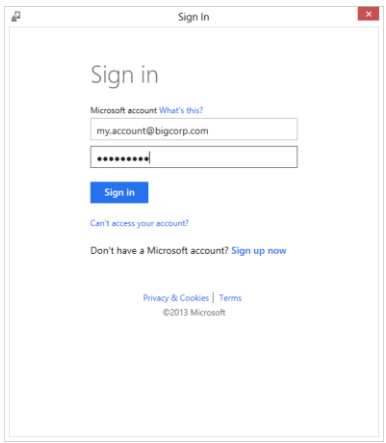


Figure 14-12Registering a Phone

The registration will take just a few seconds. Once it has completed you will see a confirmation page.

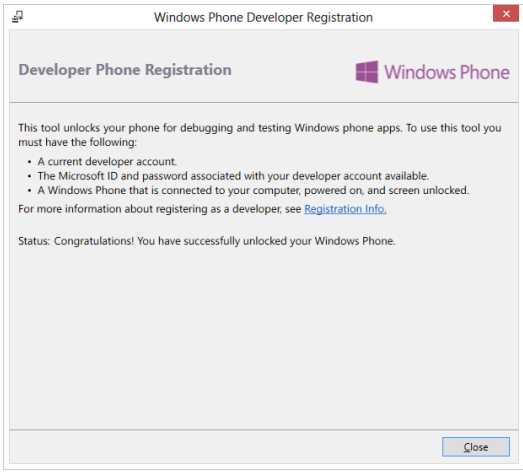


Figure 14-13Phone registration confirmation

A phone will stay registered for around a year from registration. When the registration of a phone lapses you will be unable to run any developer programs on the phone until it is re-registered. You can de-register a phone at any time using the same tool. You should do this if you ever pass your phone onto somebody else. There is a "dashboard" are in the Windows Phone developer portal where you can view and manage your developer devices.

Applications that are deployed to the phone will stay in the phone memory and can be run from the application or game hubs in the same way as ones obtained from the Marketplace. However, you can only have up to 10 of your own programs on a given device at any time.

Distributing XAP files

If you want to send a program to another developer you can send them the XAP file and they can use the Application Deployment program to send the XAP file to their unlocked device or the emulator on their Windows PC.

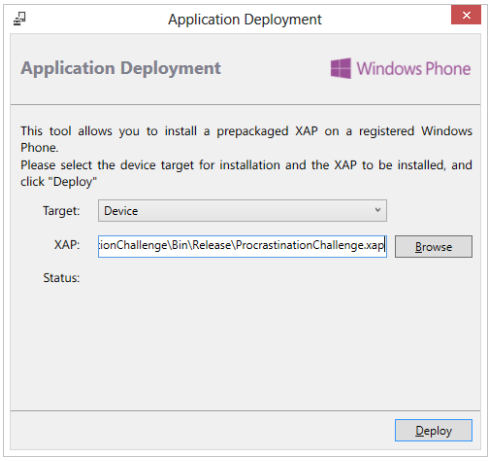


Figure 14-14Deploying a XAP file onto a registered phone

This allows you to distribute applications for testing before you release the finished product.

Program Obfuscation

If you send someone a XAP file it is very easy for them to open this up and take a look at all things in it, including all the program code that you spend so long writing. Unfortunately it is a simple matter to open an assembly and take a look at how it works. We did this in chapter 3 using the ildasm program. This means that you should be careful when sending out programs that you don't give away all the hard work that you put into making the program. Whenever you send out a program you should take steps to make it more difficult for someone to unpick the contents.

In software development this process is called *obfuscation*. It involves the use of a special tool that you use in the build process to make it very hard for someone looking at your program to deduce how it works and what it does. The tool changes variable names and adds spurious control flow to make a program very hard to understand.

There are a number of free obfuscation tools that you can use to make your program harder to understand. You could use these on your program code before you submit the XAP file. The Windows Phone developer programme also includes access to obfuscation tools from PreEmptive Solutions. These are being provided at very low cost (these systems are usually extremely expensive) and you should consider using them if you are concerned about this issue. These tools can also be used to instrument your code and allow you to find out what use is being made of the different parts of your program.

14.3 Windows Phone Store

We now know enough to make complete applications and games that will work correctly within the Phone environment and use the built-in features of the phone system. We are now going to take a look at how we can make sure that our programs are ready for the store, then we are going to look at the submission procedure and finally we are going to consider some things we can do to make our programs stand out, and hopefully increase sales.

The only way that a Windows Phone owner can get a program onto their device is by downloading it from the Windows Marketplace. Only programs that have been through the Marketplace validation process can be loaded onto a phone.

Obtaining Windows Phone Applications

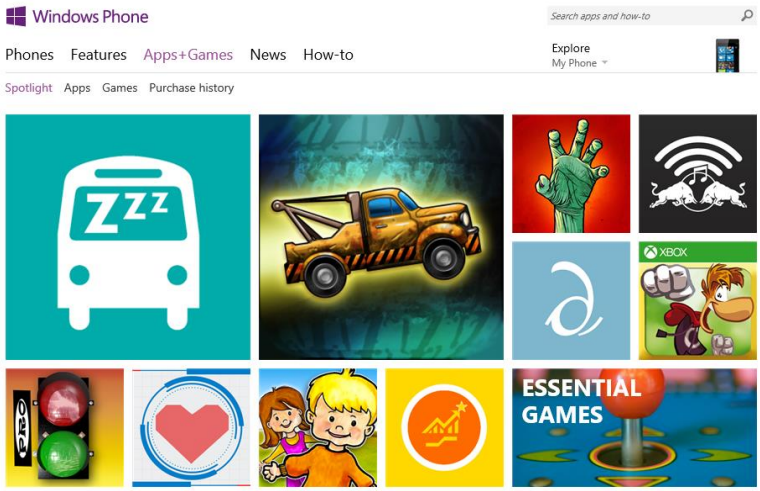


Figure 14-15Windows Phone Store

A Windows Phone owner can use their Windows Live ID to sign into their phone and gain access to services provided by Windows Live, Zune and Xbox Live along with the Store.

Windows Phone owners can download applications via the Marketplace application on the device (via WIFI or the cellular network), the Zune software on the Windows PC or from the Windows Phone Marketplace tab on the Windows Phone website `www.windowsphone.com`. Very large applications cannot be downloaded via the cellular network and must be transferred using WIFI or the Zune program. You can purchase apps using the web interface and have them automatically downloaded into the phone.

A given Windows Live ID can be used on up to five phone devices. If you remove an application from a device you can reload it later without buying it again as the Marketplace keeps a list of everything you have bought. The marketplace also provides a rating system which allows owners of applications to give ratings and feedback on them.

Creating Windows Phone Applications and Games

A registered Windows Phone developer can upload their applications and games to the Store for distribution. They do this via the “Dev Center” at `dev.windowsphone.com`

Figure 14-16 Windows Phone Dev Center

The Dev Center is also a very useful place if you are just starting out developing. There are lots of sample programs, videos and How-Tos that will get you started.

Marketplace Approval Testing

Before an application can be distributed by the Marketplace it must first go through an approval process. This makes sure that the application behaves correctly in respect of things like the Start and Back buttons and also that the program doesn't do anything silly like grab all the memory or take twenty minutes to start running. This process also makes sure that programs that are offensive are not made generally available.

The testing is partly automatic and partly human. An engineer will run your program and ensure that it works correctly. The application will be tested with light and dark backgrounds on the phone to make sure that all the menus are visible.

If your application is not approved you will receive a written report that gives details of the parts of the program that need attention. When you resubmit the re-test will focus on the areas that were identified, not on the whole program again. Sometimes an application will be approved with some "warnings". This means that issues have been identified that need to be addressed, but the situation is not so serious as to require that the application cannot be approved. In that situation you would be expected to release an upgraded version later.

Once an application has been approved it is listed in one of the application categories and is available for download. A developer can produce an upgraded version of an application which, once it has been through the approvals process, will be made available as an upgrade to all existing owners.

Some applications are free and others are paid. When a phone owner buys a paid application the cost can be added to their phone bill or they can register a credit card to be used to pay for purchases. Many applications have a "trial" mode which is free. This can later be upgraded to a fully featured version. When a program is running it can easily test whether it is being used in "full" or "trial" mode.

Registered developers are allowed 100 submissions of free applications per year. If each of these is approved this means that they can make 100 free applications available. If a developer wants to submit any more free applications each submission will cost them $20. A developer can submit an unlimited number of "paid" applications.

Applications can be loaded direction onto the phone "over the air" using the mobile phone operator or Wi-Fi. They can also be loaded using the Zune software that also provides a media management for the phone as well. Marketplace membership

Before a developer can submit applications to the Marketplace they must first become a member. Membership is keyed to the Windows Live ID of the member. It costs $99 per year to be member of the marketplace. If you stop being a member of the marketplace all your applications are removed. Students who have access to the DreamSpark programme can get free membership of the Windows Phone marketplace.

Members of the marketplace have their identity validated when they join and the Marketplace gives them a unique key which is used to sign their applications. The Marketplace also retains their bank and tax details so that they can be paid for application sales.

Members of the Marketplace can set the selling price of their application and receive 70% of the cost paid by the customer. This is paid directly into their bank account once they have reached a threshold of $200 worth of sales.

Trial Mode

An application or game can be downloaded as a free demo. A program can easily determine whether or not it is operating in demo mode:

```
LicenseInformation info = new LicenseInformation();

if ( info.IsTrial() )
{
    // running in trial mode
}
```

The `LicenseInformation` class is in the `Windows.Phone.Marketplace` namespace. If the program is running in trial mode it can restrict the user actions, for example it could disable certain functions or stop the program running after a time limit. If the user wishes to upgrade the application it can use the Launcher mechanism to direct the user to the marketplace.

However, you should remember that a paid for program with a trial mode will still only be visible in the "paid" application part of Windows Phone Marketplace. This means that customers who only ever look for free applications will never see your program. Perhaps a more successful strategy might be to produce two versions of your program, a limited "free" one and a fully featured "paid" version. That way potential customers have more chance of finding and using your program.

In-App Purchase

You can allow the user to purchase features while they use your application or game. The "in-app" purchases can be

Adding Advertising

If you don't want to sell your application, but you do want to make money from it, you can add in-application advertisements. These can be added to applications and will display advertising content to users. The user can respond to the advertisement by linking through to the advertisers web site or placing a phone call to them. In either case you will receive 70% of the advertising revenue that is generated.

Figure 14-17Advertisments in Applications

The advertisements are displayed in particular areas of the phone screen. You can include advertising in applications in a large number of regions around the world. The Advertising SDK is part of the Windows Phone SDK and provides a set of test advert servers that you can use in applications.

The Submission and approval process

The submission process for programs is not hard to use. It is all managed from the developer site for Windows Phone and XNA Games:

```
http://dev.windowsphone.com
```

Visitors to the site can download documentation and the development tools. Members of the Marketplace can use the dashboard pages to view the progress of submissions and submit new programs. There are a number of detailed walkthroughs that will take you through the membership and submission process, you should go through these to make sure you understand what is going on and what is required of you and any programs that you submit.

Windows Phone Application Certification Guidelines

The Windows Phone developer website provides access to a very detailed document that describes how to create applications. It is very important that you read the latest version of this document and follow the guidelines set out in it. It has been through numerous versions as the process has evolved, so you should make sure that you are using the most up to date version of the text. You can find the guidelines here:

```
msdn.microsoft.com/en-us/library/hh184844.aspx
```

The Store Test Kit

One way to make sure that an application passes the approvals process first time is to make use of the Store Test Kit. This is built into the development tools and provides access to the same set of tools and procedures used by the testers. Visual Studio will check for changes to the procedures and update the Test Kit automatically so that it always reflects approvals policy.

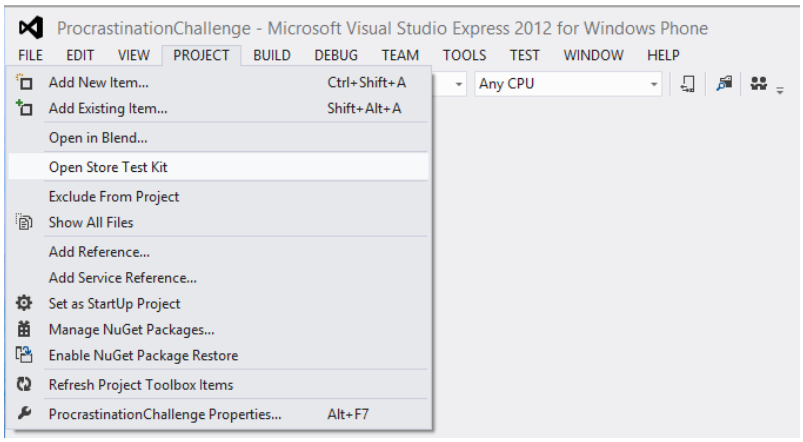


Figure 14-18 Opening the Store Test Kit

It can be found in the Project menu.

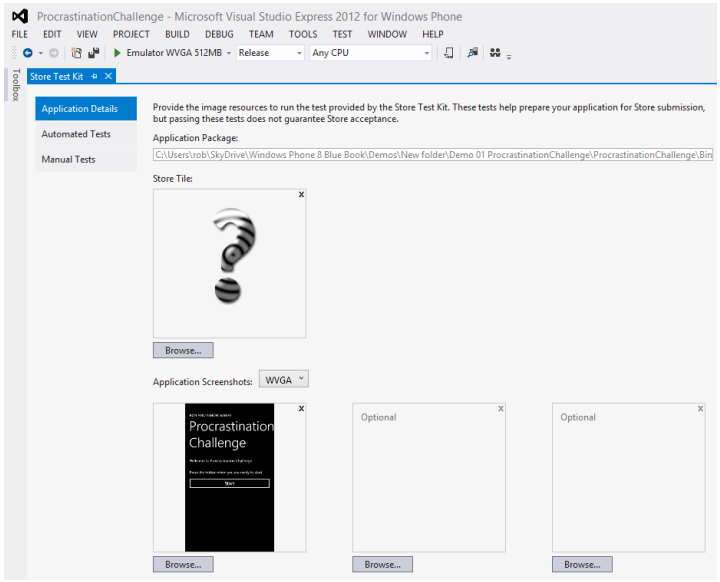


Figure 14-19 Store Test Kit Tests

The tests even check for the correct file type and size of the application tiles. There are also some automated tests and the Test Kit also provides an itemised walkthrough of the tests performed by the testers so that you can perform these tests before you submit your application.

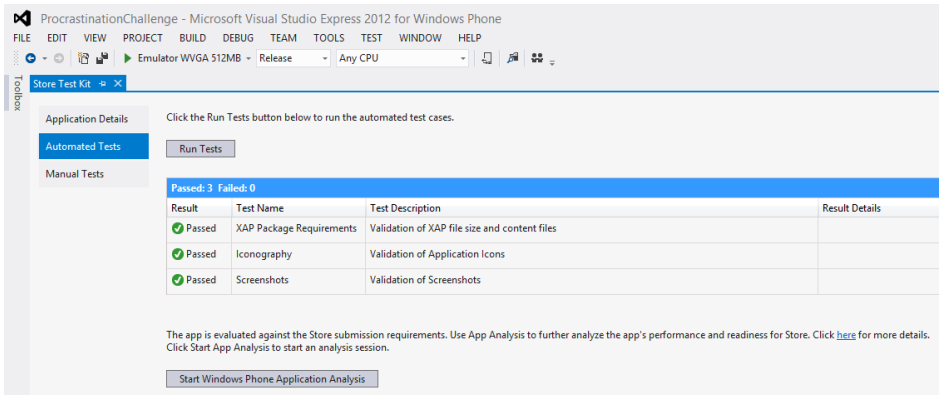


Figure 14-20 Automated Tests

This shows some that the application has all the required data and that the behaviours and capabilities all line up.

The App Submission Process

The App Submission process is initiated from the Windows Pone Dev Center web page. There are a number of stages that the application will pass through before reaching the marketplace. You initially give details and pricing information and then the application is physically tested before being released into the store.

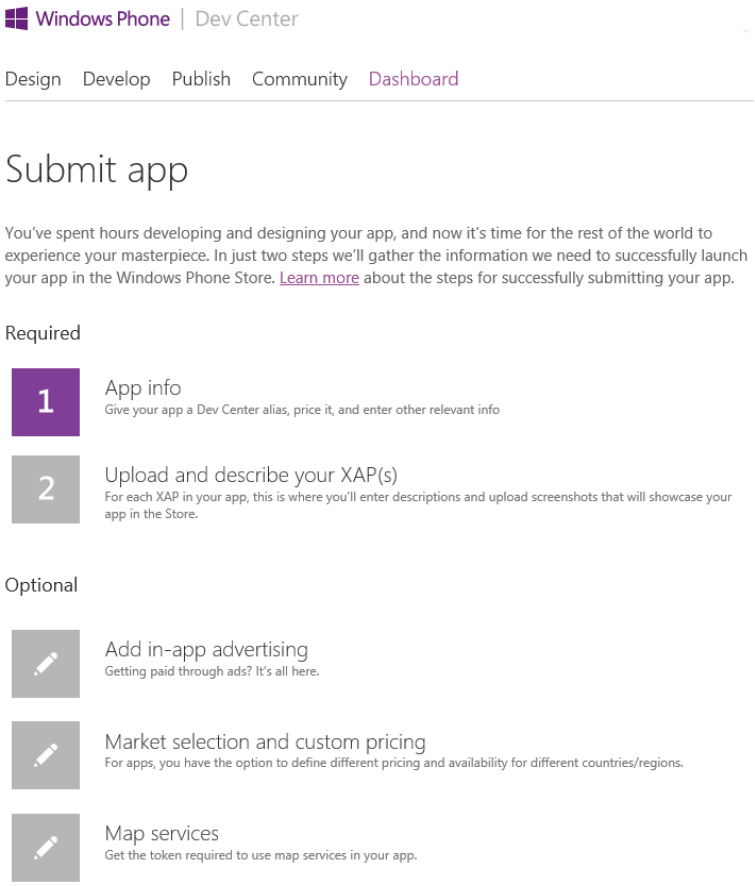


Figure 14-21 Starting the submission process

Private Beta Testing

If you want to get feedback from users before formally releasing your application you can create a "private" beta test release. Rather than making the application visible to all the customers on the marketplace, this form of release sends a weblink to up to 100 testers who can follow the link to a version of your program that can be downloaded onto a phone and used for up to 90 days, after which time the application is removed from their phone.

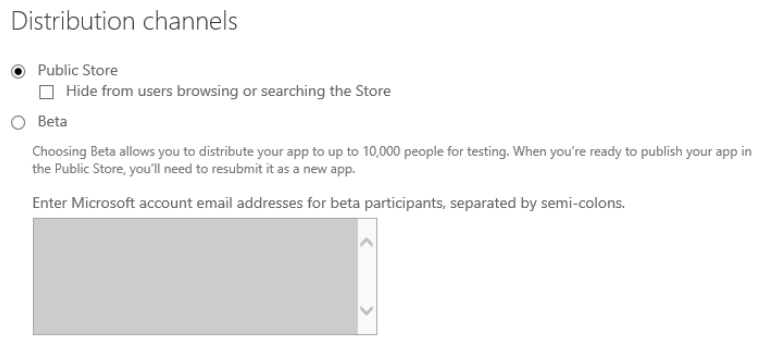


Figure 14-22 Creating a Private Beta

This option is set at the beginning of the submissions process. Members of the beta will be sent a web link they can use to download the program. You can also hide an application from casual browsers so that it too can only be installed directly from a link.

14.4 Making your Application Stand Out

The Windows Phone Marketplace is now acquiring a fairly large number of applications and so if we want our programs to be the ones that are downloaded we have to do a bit of extra work to get noticed. Here are some tips that you might find useful.

Design to Sell

An attractive looking application will have a much greater chance of being noticed. The tiles, backgrounds and screenshots that are displayed on Marketplace should all be designed to help sell the application. You should also make sure that you provide lots of screenshots and also the large background graphic. Do not do the Marketplace description just before you publish the application. Make sure that you spend some time getting the words and the display just right. A really attractive application might even be picked as a specially selected one and highlighted on the Marketplace.

Target Different Localisations

While there might be more than one application like the one that you are selling, there might not be any in the German language. So, by producing a localised version of the program you could have that marketplace to yourself. There is very good localisation support available to Windows Phone developers, you can find out more here:

```
http://msdn.microsoft.com/en-us/library/w7x1y988.aspx
```

Search Extensibility

The Search Extensibility feature in Windows Phone makes it possible for customers to find your application even when they are searching for other things. For example a Bing search for “Knitting wool” could be linked through to your knitting pattern application. The process of linking an application in this way is free and all you have to do is add details of the categories under which your application can appear. You can find out more here:

```
http://msdn.microsoft.com/en-us/library/hh202957.aspx
```

Give your Program Away

If you provide a free version there is more chance that users will download it and use it. The free version could contain adverts or “nag” messages. Because it is easy to create a launcher that will take the user into the Windows Phone Marketplace you can provide a “buy upgrade” option that can target a paid version of the program or game.

Release Upgrades/Episodes

Rather than releasing a 15 level game or a fully featured program you could instead release a “starter” version and then provide upgrades at regular intervals. There are a number of reasons why this is a good idea.

- You will get to market earlier than your competition.
- Your application will appear regularly in the "new items" list on Marketplace. Since a lot of sales of an application are made in the few days after a new release, making more new releases will improve your sales.
- You can get the customers engaged in providing ideas for new content and features. This means that you can build a good relationship with them.

Change Categories

When you submit an application you will be asked to choose the category in which it is to be displayed in the Marketplace. It is often the case that one particular category doesn't really work. A game could be thought of as a sports simulation, an arcade game or a puzzle. If you are not sure which is the best category, try them moving the game around and see how this affects sales. This will also get you exposure in other categories, which will also be beneficial.

Encourage Good Feedback

A good relationship with customers is something that you should work hard to cultivate. In the unlikely event of your application failing it should display a populated mail message and provide the user with an easy way of sending it. Then, if you receive such a message you should respond constructively. Building a relationship in this way makes customers into debuggers and advocates of your programs, which is really useful.

14.5 What To Do Next

At this point in the document you should have all the skills you need to make something that can be published on Windows Phone. Here is a list of things you can do to take the next step.

Register as a Developer

If you have not already done this you should head over to `dev.windowsphone.com` and sign up. If you are a student you should stop off at `dreamspark.com` and get a free registration.

Get the Toolkit

The tools can be found at `dev.windowsphone.com` and will provide you with everything you need in a single install. This includes Visual Studio, the emulator, Expression Blend, the Advertising SDK and programs to unlock your device.

Publish Something

Once you have got your development environment working you should put something out in Marketplace. It doesn't really matter whether it is something you are massively proud of, and you can always remove it later, but you never know, there might be someone out there desperate for the application that you make.

Make Suggestions

If you find something you don't like, or have an idea that could make things better, you can use the UserVoice forums to suggest things and vote on them. The Windows Phone team really do read and respond to these suggestions, and quite a few have found their way into the device. You can find the forums here: `wpdev.uservoice.com`

Program Ideas

We now know how to make programs for Windows Phone. We can create multi-page user interfaces which match those of the phone itself and we have also had a look at how to create games using XNA. We also know how to create connected applications that make use of data hosted on the internet. We have discovered some of the difficulties encountered by programmers when they write code for small, resource constrained, devices and we now also know how the Windows Phone operating system allows us to create useable applications in spite of these issues. Finally we know how to use the underlying Windows Phone system so that our applications can make use of features in the phone itself.

At this point we have a superb set of tools, next we have to find a problem to solve with them.

Application Ideas

Applications ideas do not always appear fully formed. Quite often you will start with a small idea for a solution and then add features or discover situations where the solution could be made even more useful. This is not a bad way to come up with something original, but you must beware of adding too many feature ideas before you have made something work, otherwise the whole thing might collapse under its own weight before you have built anything.

You can get good ideas for applications by talking to lots of people and trying to find out what would be useful to them. Not everyone is aware of just how much you can do with modern devices and so if you say that you can make something mobile that can track position, take pictures and make use of internet services they might have an idea of a situation where some of that power would be useful to them.

Game Ideas

In some ways game ideas are easier to come by than application ideas. Systems like XNA let you "doodle" with game code to find out what it does. I've already mentioned the importance of playtesting. This is where you give your game to other people to find out if the game is playable. Playtesting can also throw up ideas for game development. Often a playtester will suggest changes to the program that will make it more interesting and fun. Don't be afraid to do silly things in a game (add 10 times as many aliens, invert gravity, make everything bigger/smaller etc etc) and see what happens to the play experience.

The great thing about a game is that rather than solving a specific problem, as with an application, a game just has to be fun to play.

Fun with Windows Phone

Windows Phone provides an amount of processing power that was unavailable in the world a few years ago, let alone in everyone's pocket. It also provides an amazing level of connectivity, high performance graphics and devices such as cameras and location sensors that make it possible to build truly novel applications.

Personally I reckon that just being able to program for this device, let along sell the results, is inspiring enough. I hope you have learned enough from these notes to get yourself started on the fun you can have with this platform.

Rob Miles
September 2013

What We Have Learned

1. A Windows Phone application is represented on the device by two icon files. One is used to represent the application in the Application list on the phone and the other, larger, icon is used if the application is pinned to the Start menu on the phone. Good icons are important as they are both used on the phone and in the Windows Phone Marketplace to brand an application.
2. Applications also have a "splash screen" image that is displayed while the application starts. The default splash screen is provided as part of a new Windows Phone project but application writers are advised to customise this.
3. Windows Phone owners get their applications from the Windows Phone Marketplace.
4. Registered developers can submit applications for approval and distribution via the Marketplace.
5. Developers register and track the progress of their applications via the Windows Phone developer website.
6. Registering as a developer costs $99 a year, students can register for free via the Dreamspark programme.
7. Developers can produce free or paid applications, but are limited to submitting 100 free applications per year. Further submissions of free applications cost $20 each.
8. A registered developer can register phones for development so that they can run programs compiled in Visual Studio. A Microsoft Account holder can also register their phone for development without becoming a registered developer.
9. Applications are distributed as XAP files which contain all the resources and content along with a manifest file that describes the content and gives the phone setting information about the application.
10. XAP files can be loaded directly onto devices that have been registered for development.

11. Programmers should obfuscate their programs so that it is not easy for anyone obtaining a XAP file to find out how the code works. There is an obfuscation processor available via the Windows Phone developer web site.
12. A developer must read the Windows Phone Certification guidelines before submitting programs to the approval process.