Microsoft

Begin to Code with C#

Rob Miles

Part 4

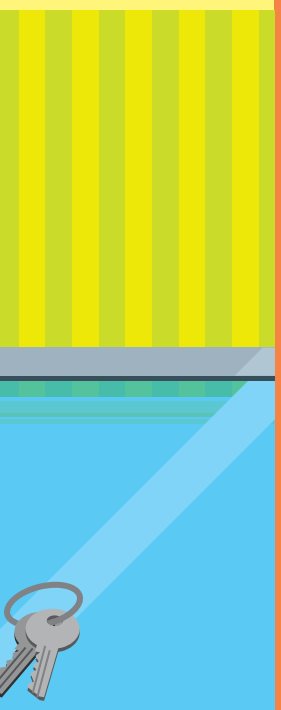
Creating applications

In Part 4, you are going take another huge step on the road to becoming a full-fledged developer. By the end of this part, you will be creating programs that you can go on to sell. Remember that the only difference between our examples and "real" programs is that someone pays money for the real ones.

At this point, we are going to start moving away from the Snaps framework, which has been hiding some of the underlying complexity of Windows 10 program development. But we will not be discarding Snaps entirely. You'll find out how Snaps elements work and how you can use their abilities in programs that you write. You are going to start by learning about designing user interfaces, and then you'll move on to consider how modern programs deal with external events.

16

Creating a user interface using objects



What you will learn

The user interface is a program's shop window. In the same way that an inviting window display can entice you into a store to buy products, a well-designed user interface can do a lot to encourage users to engage with a piece of software. In this chapter, we are going to consider how modern application user interfaces are created. You'll find out how you use software elements to represent the items a user interacts with. Then you are going to discover how programs interact with these elements and how these elements respond to actions performed by the user. Throughout the chapter, we will work with a language named XAML (Extensible Application Markup Language; pronounced "zamel" to rhyme with "camel"), which can be used to describe the design of a user interface. You'll see how you can use XAML with Visual Studio to build a great experience for users

Making an adding machine

Given the success of the talking times-table tutor we created in Chapter 6, you've been asked to develop an adding machine that people can use to practice addition or in other ways. What your friends tell you they want is something like what's shown in **Figure 16-1**. Users enter numbers into the two top boxes, press the **Equals** button, and see the results.

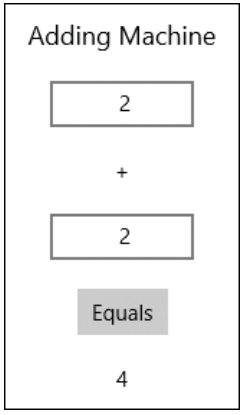


Figure 16-1 Adding machine tutor.

This is a simple Windows application, but it is actually beyond what we can do with Snaps. The programs that we have written up until now used the behaviors from the Snaps library to interact with users. Snaps work very well, but they are not very flexible. To create an adding machine tutor, we need to discover how to create graphical user interfaces (a GUI).

Toward a graphical user interface using XAML

A user interface is a posh name for what people see when they use your program. It comprises the buttons, text fields, labels, and pictures that users work with to get their job done. Part of the job of the programmer is to create this front end and then put the appropriate behaviors behind the screen display to allow users to drive the program and get what they want from it.

This section is not as much about programming with C# as it is about how to create user interfaces using XAML, which was designed by Microsoft to make it easy to create a good-looking application. You use XAML to describe what is displayed and

C# to provide the behaviors. (You can create programs that use a XAML-based user interface with other programming languages as well—Visual Basic, for example.) To understand this process, you are going to have to learn a few things about how markup languages work, but this knowledge is extremely useful because lots of modern user-interface systems work in a similar way.

An extensible application markup language lets you use the rules of the language to create constructions that can describe essentially anything. English is a lot like this. There are letters and punctuation that are the symbols of English. We also have rules (grammar) that set out how to make up words and sentences, and we have different kinds of words—nouns that describe things and verbs that describe actions. When something new comes along, we invent a whole new set of words to describe it. Someone had to come up with the word *computer* when the computer was invented, along with phrases like *boot up*, *system crash*, *too slow,* and *deleted all my work*.

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. These are called *markup languages* because they can be used to describe the arrangement of items on a page. The word *markup* was originally used in printing, when you wanted to say something like "Print the name Rob Miles in a very large font." The most famous markup language is probably HTML (for Hypertext Markup Language), which is used on the World Wide Web to describe the format of webpages. Programmers frequently invent their own data storage formats using XML. XAML takes the rules of an extensible markup language and uses them to create a language that describes components to be displayed on a page.

Take a look at the XAML description of a text box:

```
<TextBox Name="firstNumberTextBox" Width="100" Margin="4" TextAlignment="Center">
</TextBox>
```

You can see that the designers of XAML created words with meanings that match their requirements. You can give the `TextBox` element a name, it must have a particular width and margin, and you want to specify how the text in the box is to be aligned.

XAML and page design

When you use Visual Studio to create a brand-new Universal Windows application, you get a page that contains just a few elements. As you put more elements on the page, the file grows as each description is added. Some elements, like a text box, can stand alone. Other elements work as containers. This means that they can hold other elements. Containers are very useful when you want to lay out a page. For example, the `StackPanel` element can hold a set of other elements in a stacked arrangement. A

XAML file can also contain the descriptions of animations and transitions that can be applied to items on the page to make more impressive user interfaces.

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 language. But it turns out that many programmers (including me) are not that good at designing attractive user interfaces (although you might be). In real life, a company often employs graphic designers who create artistic-looking front ends. The role of the programmer is to put the code behind these displays to get the required job done.

XAML was developed with this issue in mind. It enforces a strong separation between the screen-display design and the code that is controlled by it. This makes it easy for a programmer to create an initial user interface that is subsequently changed by a designer to become a much more attractive one. And it is also possible for a programmer to take a complete user interface design and then fit the behaviors to each of the display components.

Describing XAML elements

We'll start to discover how XAML lets us design an application by using it to create our adding machine. Look back at **Figure 16-1** to recall what the user interface will look like. This display is made up of six components:

1. The title Adding Machine. This block of text is slightly larger than the rest of the text so that it stands out.
2. The top text box, in which a user enters a number.
3. A text item holding the character +.
4. The bottom text box, in which a second number is entered.
5. A button that a user presses to perform the addition.
6. A result text box that changes to show the result when the button is pressed. (At the moment, this text box is empty because we haven't done any sums yet.)

In XAML terms, each individual item on the screen is called a `UIElement` (or user interface element). I'm going to call these items *elements* from now on. Each elements has a particular position on the screen, a particular size for its label, and lots of other properties, too. For example, you can change the color of the text in a text box, whether it is aligned to the left, right, or center, by updating the XAML that describes the page. The XAML I used to describe the display for the adding machine is as follows:

```
<StackPanel>
  <TextBlock Text="Adding Machine" TextAlignment="Center" Margin="8"
      FontSize="16"></TextBlock>
  <TextBox Name="firstNumberTextBox" Width="100" Margin="8" TextAlignment="Center">
      </TextBox>
  <TextBlock Text="+" TextAlignment="Center" Margin="8"></TextBlock>
  <TextBox Name="secondNumberTextBox" Width="100" Margin="8"
      TextAlignment="Center"></TextBox>
  <Button Content="Equals" Name="equalsButton" HorizontalAlignment="Center"
      Margin="8"></Button>
  <TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="8">
      </TextBlock>
</StackPanel>
```

CODE ANALYSIS

Investigating XAML

If you look carefully, you can map each of the elements shown in **Figure 16-1** and listed above to items in the XAML file. But there are some elements that we should consider in more detail.

Question: What does the `StackPanel` element do?

> **Answer:** The `StackPanel` is very simple, and very useful. A `StackPanel` lets us arrange a series of display elements in a stack—which means you don't need to define the position on the screen of each of the elements individually. The default arrangement is to stack the items down the screen, but you can also stack items across the screen. You can—and this is really useful—put a `StackPanel` inside a `StackPanel` to make up a stack of rows. The nesting of elements is a recurring theme inside XAML documents.

Question: What is the difference between a `TextBox` and a `TextBlock`?

> **Answer:** A `TextBox` is a display element in which a user can type text. The two numbers that are going to be added will be entered into the `TextBox`es named `firstNumberTextBox` and `secondNumberTextBox`. A `TextBlock` is simply a block of text that is displayed. We use the `TextBlock` to tell the user things. In this case, we use `TextBlock`s to show the title of the application, the + sign for the addition, and the actual result of the calculation. The user can't interact with the content of a `TextBlock`.

Question: Just to check that you understand how this works, what is the text presently being displayed in the `resultTextBlock`?

> **Answer:** You can work this out by looking through the XAML and finding the `TextBlock` with the name `resultTextBlock` and looking for the `Text` property of that `TextBlock`.

It turns out that when the program starts, the `Text` property is set to "", or an empty string.

PROGRAMMER'S POINT

Use automatic layout as much as you can

I always worry when I start positioning elements on the screen in absolute positions. As soon as you do this, you are making assumptions about the size of the screen that you are using and the dimensions of the element. Modern computers are supplied in a huge range of different screen sizes, and users can also change the size of the text on the screen to zoom into the display. Users might also change the orientation of their screen from landscape to portrait while using your program. If you fix the position of things on the screen, this means that it may work well for one particular device, but it will look very awkward on another. For this reason, you should use automatic layout features such as a `StackPanel` to dynamically position things for you. This makes your program much less likely to have display problems.

XAML elements and software objects

From a programming point of view, each XAML element on the screen is actually a software object in the program. You have already seen that 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 its position on the screen, the color of the text, the text itself, and so on.

When a program that uses a XAML user interface is compiled, the system also "compiles" the XAML description to create a set of C# objects, each of which represents a user-interface element. There are three different types of element in the adding machine tutor:

- **TextBox** Allows the user to enter text into the program.
- **TextBlock** A block of text that just conveys information.
- **Button** Something we can press to cause events in our program.

Our program will manipulate these elements as though they are C# objects, although they are actually defined in a XAML source file. This works because when the program is built, the XAML system will create objects that match elements described in the XAML source file.

Managing elements by their names

When you want to use the user interface elements in a program, you need a way of referring to them. If you take a look at the XAML that describes the adding machine tutor, you can see that some of the components have a name property:

```
<TextBox Name="firstNumberTextBox" Width="100" Margin="4" TextAlignment="Center">
    </TextBox>
```

I gave this text box the name `firstNumberTextBox`. (You'll never guess what the second text box is called.) Note that the name of an element in this context is actually going to define the name of a `TextBox` variable that's declared inside the adding machine program. In other words, there will now be the following statement in the program somewhere:

```
TextBox firstNumberTextBox;
```

These declarations are created by Visual Studio when the program is built, so you don't need to worry about where this statement is. You just have to remember that this is how the program works.

CODE ANALYSIS

XAML element names

Question: Why don't all the display elements have names?

> **Answer:** We only need to give names to elements that our program needs to interact with. There's no point in giving a name to the `TextBlock` that holds the "+" character because users will never need to interact with this when the program runs—it just holds something that will be displayed for the user. Of course, if you want to change this element later (perhaps to make it possible for the user to select a version of the program that will perform subtraction), you can give it a name.

Properties in elements

Some of the properties of a `TextBlock` are set in the XAML that declares it. Others must be changed by the program as it runs. All the properties can be set in the declaration and then changed by software as well.

Here is the XAML that describes the part of the screen where the result of the addition is displayed:

```
<TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="4">
    </TextBlock>
```

Note that there is a `Text` property that is currently set to an empty string. When we talk about the "properties" of XAML elements on the page, we are actually talking about property values in the class that implements the `TextBlock`. In other words, suppose a program contains a statement such as this:

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

This statement would cause the text 0 to appear inside the `resultTextBlock` on the display. It will also cause a `Set` property assignment to run inside the `resultTextBlock` object, which sets the text on the `TextBlock` to the appropriate value. In other words, the `Text` property of the `TextBlock` is the same property, whether you set its value in XAML or within a C# program.

Page design with XAML

XAML turns out to be very useful. Once you get the hang of the information needed to describe components, it becomes much quicker to add elements to a page and move them about by just editing the text in the XAML file rather than dragging text boxes and other elements around the screen. 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 provides IntelliSense support as you go along.

If you read more about the XAML specification, you'll find that you can give elements graphical properties that can make them transparent, add images to their backgrounds. and even animate them. At this point, you have moved beyond programming and entered the realm of graphical design—and I wish you the best of luck.

Now that you know that items on the screen are in fact the graphical realization of software objects, the next thing you need to know is how to get control of these objects and make them do useful things in an application. To do this you need to add some C# code that will perform the calculation that the adding machine needs. But first let's build the application itself.

MAKE SOMETHING HAPPEN

Build our first Universal Windows Application

This is the point at which we build our first Universal Windows Application. Up until now, we have built everything inside the Snaps environment. Now we are about to create a brand-new, completely empty application. This is the same step that every application developer takes when he or she decides to build a new program.

Creating a new application

To begin, start Visual Studio 2015. Once Visual Studio is running, click the **File** tab, move to **New**, and then click **Project**, as shown in **Figure 16-2**, to open the **New Project** dialog box.

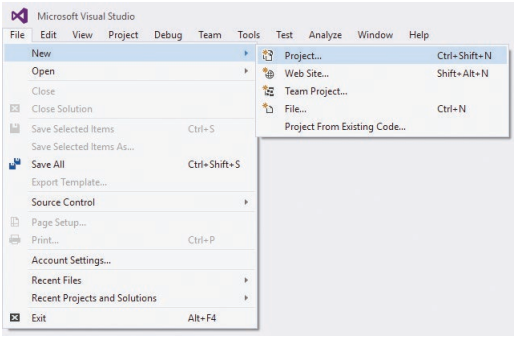


Figure 16-2 Visual Studio New Project menu.

There are lots of different kinds of projects that you can create. We want to choose **Blank App (Universal Windows)**. On the left of **Figure 16-3,** you can see how to navigate to **Templates>Visual C#>Windows>Universal** to get to this set of project templates.

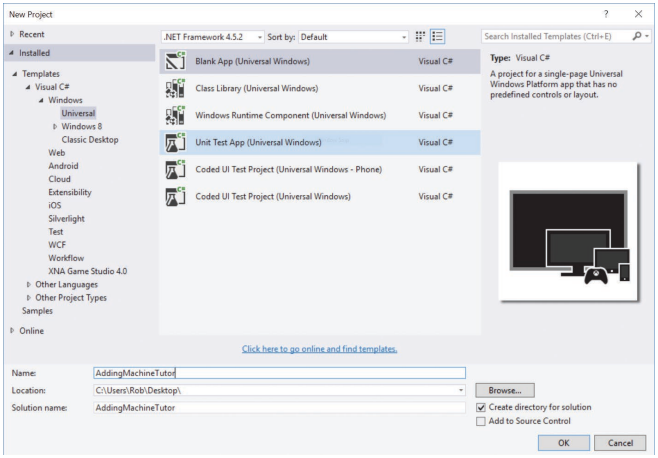


Figure 16-3 Naming our new AddingMachineTutor project.

During the solution-creation process, you might be asked which versions of Windows you want the application to work with, as shown in **Figure 16-4**. Just select OK to use the default versions.

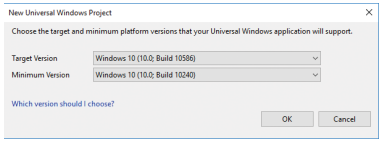


Figure 16-4 Keep the default settings when prompted for which versions of Windows to use.

WHAT COULD GO WRONG

You don't see Universal Windows Application templates

If you can't find any Universal Windows templates in the **New Project** dialog box, you may be using the wrong version of Visual Studio. You must be using Visual Studio 2015. If you're using Visual Studio 2015 but you still can't see this project type, make sure that you have the Universal Tools installed for Visual Studio 2015. These are usually installed as part of the overall installation of Visual Studio 2015, but you may have an older installation that doesn't have them. Take a look at the instructions online (see the link in Chapter 1) that describe how to fix this.

Once you find the required project type, enter a name for this project (I called my project **AddingMachineTutor**), and then click OK. By default, Visual Studio will create

the new solution in a subfolder of your Documents folder. You can change where the solution is created by clicking **Browse** and navigating to a different location on your machine before you click OK.

WHAT COULD GO WRONG

Picking the wrong template never ends well

It's kind of embarrassing to admit this, but in the past I've been known to pick the wrong template at the start of my projects. I usually do this during a demonstration to at least 200 students. This results in a lot of confusion for me, and much amusement for them, so I'd advise you to check very carefully that you select the right one unless you want to look as silly as I do.

Creating an empty program

After you click OK, Visual Studio 2015 goes to work and creates a new empty project for you. **Figure 16-5** shows you what Visual Studio displays after it has created a new application.

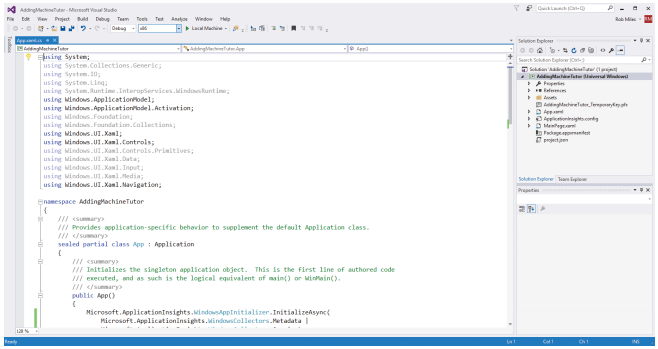


Figure 16-5 An empty Universal Windows Application.

This looks very confusing, and it is. Visual Studio is showing you the contents of a file in your application named **App.xaml.cs.** This is an important file—it is the part of the application that gets control when the program starts running—but for now we can leave it as it is. (We can always reopen it later if we need to make any changes.) To close the view of the file **App.xaml.cs**, click the **X** next to the file's name, as shown in **Figure 16-6**.

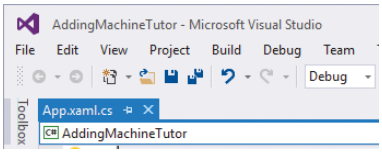


Figure 16-6 Closing a window.

We can run our empty application in the same way as we have been running Snaps applications. Click the run button (the green arrow) on the top row of controls (being sure that the text next to it shows **Local Machine**). When you click the run button, the program is compiled, loaded into memory, and then allowed to run. **Figure 16-7** shows the result.

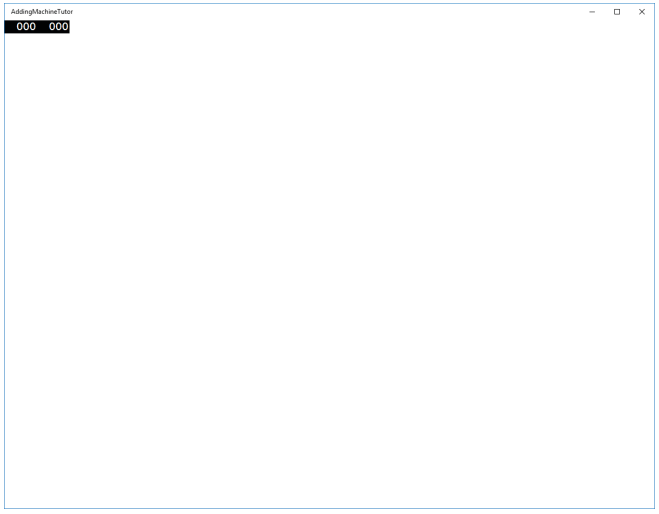


Figure 16-7 Running an empty program.

The empty program looks pretty much as you would expect, although you might be curious about the two numbers in the top-left corner of the application window. These are performance counters that tell you the demands your application is placing on the host computer and the rate that the display is being updated. They are not particularly important just at the moment, so you can ignore them for now.

Bearing in mind that all we have done is make an empty program, we do have a lot of functionality. We can drag the window around the screen, change its size, maximize and minimize it, and close the application down. We could even submit the solution to be sold on the Windows Store, although it is unlikely it would be approved for sale because it currently doesn't do anything.

CODE ANALYSIS

Stopping a Windows application

Question: How do we stop a Windows application?

> **Answer:** In the programs that we have written up until now, the `StartProgram` method from Snaps has been called at the start of the program, and when this method finishes, the program ends. A Windows 10 application doesn't work like this, however. The program will "run" until the user closes it or the computer is switched off.

Of course, a Windows 10 application is not really running in the same sense as our earlier programs, in that most of the time it is asleep until the user actually does something. Our Adding Machine Tutor program will spend most of its time waiting for the user to type in some numbers or press the button to trigger the calculation behavior.

If you need a program to get control when the user closes the program—perhaps to save some data—you can connect a method to the event that occurs when the user closes the program.

Creating the user interface using XAML

The next step is to add some elements to the user interface. We'll do this by adding some content to the XAML file that describes the page. The XAML that describes the main page of the application is held in a file named **MainPage.xaml.** First, you need to stop the application if it still running. Then open **MainPage.xaml** by double-clicking it in Solution Explorer, as shown in **Figure 16-8**.

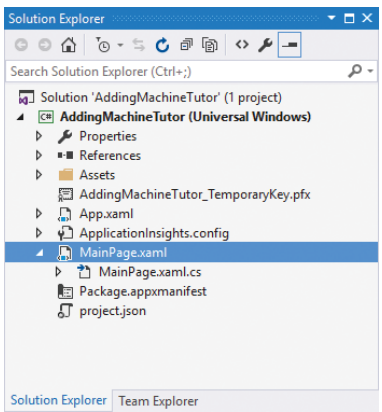


Figure 16-8 Opening the MainPage.xaml file.

Opening the file displays it in an editing view that lets you see the XAML that defines the design along with a preview of how the page will look, as you can see in **Figure 16-9**.

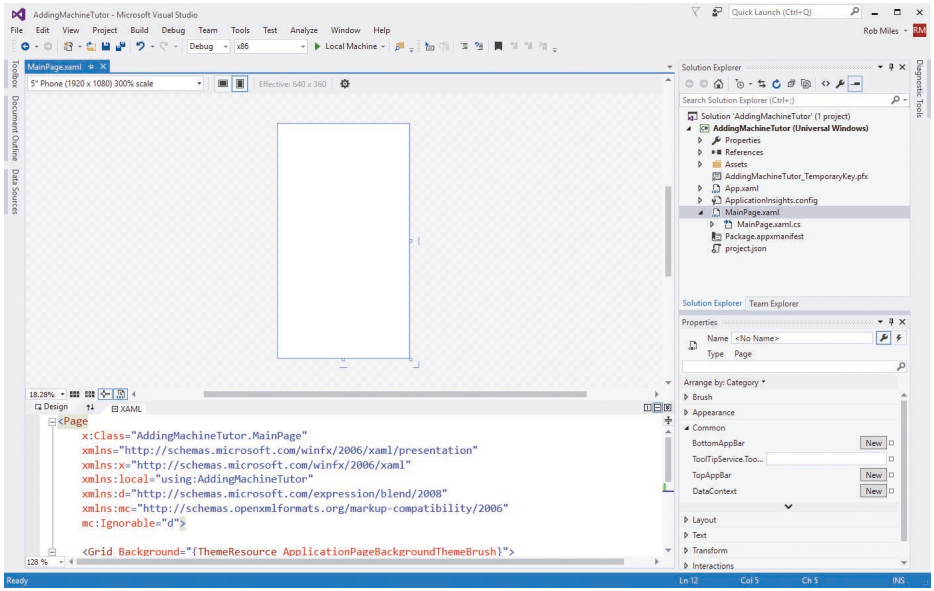


Figure 16-9 Editing XAML in Visual Studio.

The editing area is split into two regions. At the top you have a preview of the interface as the user will see it. At the bottom you have the XAML that describes the elements on the page. We are going to start by editing the XAML file.

At the very bottom of the XAML file is the `Grid` element that contains the elements that appear on the screen. You can see the top row of the `Grid` description at the bottom of **Figure 16-9**. We can add an element to the screen by adding XAML to the `Grid`:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel>
        <TextBox Name="firstNumberTextBox" Width="100" Margin="8"
            TextAlignment="Center"></TextBox>
    </StackPanel>
</Grid>
```

This is the definition of the first `TextBox` on the display. If we run the application again, the `TextBox` is displayed at the top of the application, as shown in **Figure 16-10**.

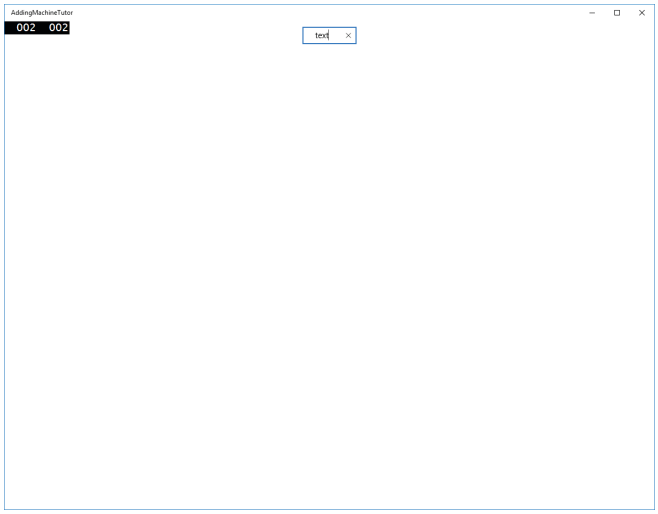


Figure 16-10 A single `TextBox` on the screen.

You can enter text into the `TextBox`. The text you enter is centered inside the `TextBox` because the `TextAlignment` property for the `TextBox` has been set to `Center`. Now go ahead and add the other elements to the display at this point.

```
<StackPanel>
    <TextBlock Text="Adding Machine" TextAlignment="Center" Margin="8"
        FontSize="16"></TextBlock>
    <TextBox Name="firstNumberTextBox" Width="100" Margin="8"
        TextAlignment="Center"></TextBox>
    <TextBlock Text="+" TextAlignment="Center" Margin="8"></TextBlock>
    <TextBox Name="secondNumberTextBox" Width="100" Margin="8"
        TextAlignment="Center"></TextBox>
    <Button Content="Equals" Name="equalsButton" HorizontalAlignment="Center"
        Margin="8"></Button>
    <TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="8">
       </TextBlock>
</StackPanel>
```

These XAML statements define the elements that make up the adding machine tutor display. This XAML looks a bit like a program, but it is actually quite different. It is a declaration of all the display elements that the program will be using. Visual Studio will work through this file and create the display elements when the program is compiled. If you enter this text into **MainPage.xaml**, you can see the elements appear in

the preview window above the text. If you run the program, you will see the elements displayed on the screen as you have specified, which you can see in **Figure 16-11**.

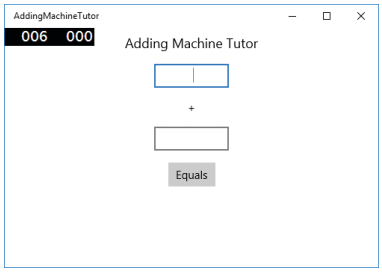


Figure 16-11 Adding machine tutor.

If you change the size of the program window by dragging a corner of it on the screen, you'll notice that the elements always stay in the center. Windows will automatically update the layout if the size of the enclosing window changes. The nice thing about this kind of design is that it will display correctly on any device, from an 84-inch Surface Hub screen to a much smaller Windows Phone.

CODE ANALYSIS

Positioning elements on a XAML display

I've been advising you to use storyboards to set out what a program should look like, and XAML provides a great way to take storyboard designs and build them into the application. However, when you are planning your designs, you need to have an understanding of how dimensions are specified within XAML.

Question: How do we measure things within a XAML design?

Answer: Simple question, complicated answer. In the old days, it was easy to express dimensions on displays. We just measured everything in pixels. (Pixel is an abbreviation of "picture element" and is a single addressable dot on a display.) A dimension of 100 meant 100 actual dots on the screen. This approach worked well because screens were low resolution and there weren't many different sizes in use. In addition, a program would only be expected to work on one platform.

Nowadays, there are many different sizes of display, from tiny tablets to enormous wall-mounted screens. And just to make things worse, the displays themselves vary greatly in the number of pixels they contain. Some devices have LCD panels with hundreds of pixels per inch; others make do with a much lower resolution. Sizing everything in pixels no longer works.

The dimensions used in Universal Windows Applications have been created to make applications portable across different screens. When we considered the size of sprites in our games, we observed that the "pixel" values you use in XAML are scaled by Windows to reflect the underlying hardware, so that 96 of them are equivalent to one inch on the display. But it's slightly more complicated than this. The pixel values, called *effective pixels,* are actually scaled to take account of viewing distance, display size, and display resolution so that they "look right." This means that something specified as 100 pixels wide may be drawn using 150, 175, or 200 physical pixels, depending on the target device.

In addition to effective pixels, Windows 10 also provides a feature called *Adaptive User Interface*, which lets you create alternative designs for different display sizes. The idea is that when your program runs, it automatically picks the design that works on the display in use. On a large screen, the user will see multiple panels, and on a smaller screen a single panel design will be used, with the user navigating between them.

Question: Is the `firstNumberTextBox` in the center of the screen because I've set the property `TextAlignment="Center"` on it?

> **Answer:** No. The `TextAlignment` property refers to the alignment of the text inside a `TextBox`. Setting this property to `Center` means that when a user enters text into the `TextBox`, the cursor will be placed in the center of that box. It turns out that, unless you specify otherwise, elements are naturally centered by XAML. However, you can add a property to an element to tell it to position itself differently. For example, if you add `HorizontalAlignment="Left"` to the `firstNumberTextBox` element, it will move itself to the left edge of the screen. There are lots of interesting properties that you can set for elements. You can find out about them by using the IntelliSense feature of the XAML editor, which will suggest the properties and also the values that they can have.

Previewing XAML screen display sizes

If you look at the preview screen in Visual Studio and the display you get when the application runs, you'll notice that the pages are completely different shapes and sizes. The `TextBox` looks bigger on the preview screen in Visual Studio, too. Visual Studio provides a range of preview environments with different sizes. You can see them if you open the combo-box at the top-left corner of the XAML editor. **Figure 16-12** shows the device types that are preset in Visual Studio.

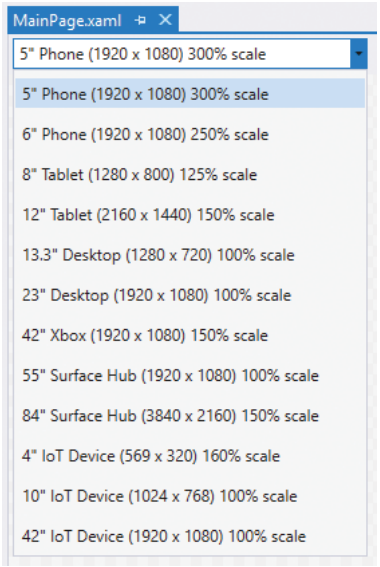


Figure 16-12 Display size options in XAML.

I'd really love to write a program that displays on the 84-inch Surface Hub. Or one that runs on the Xbox One. If you have a particular device in mind for your application, you can select it so that the preview in the editor reflects this.

Note that to the right of this list is some useful information about the preview window in use. As you can see in **Figure 16-13**, you can read off the effective number of pixels available on the screen setting that you have selected for the preview page. This means that if I created a `TextBox` that was 360 pixels wide, it would spread across the entire screen.

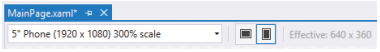


Figure 16-13 Effective sizes in XAML.

PROGRAMMER'S POINT

Design your user interface for maximum flexibility

This material on sizes and design is beyond the true scope of our programing book, but I think there is one very important point to remember here—and that is that fixed dimensions and positions are not really your friends. You can use the designer in Visual Studio to precisely place items on the screen, but I think that this is a very bad idea in a world with so many different device formats available. If you position things on the screen using too many fixed values, you are going to have a very inflexible display. At some point you are

going to be faced with an irate user who is complaining that they can't see the **Submit** button when they use your program on their particular tablet PC. I make use of the `StackPanel` container to lay out simple items, and I suggest that you do the same.

Adding the program behaviors

You can use XAML to create rich and interesting user interfaces, but none of them will actually do anything for your users. To get things done, you need to run some program code. Whenever Visual Studio makes a XAML file that describes a page on the display, it also makes a C# program file, called the *code-behind* file, to go with it, and this is where we can put code that will make our application work. The C# program file for a particular XAML page is located under its entry in Solution Explorer. You can open it by clicking the arrow to the left of the page name and then double-clicking the name of the source file, as shown in **Figure 16-14**.

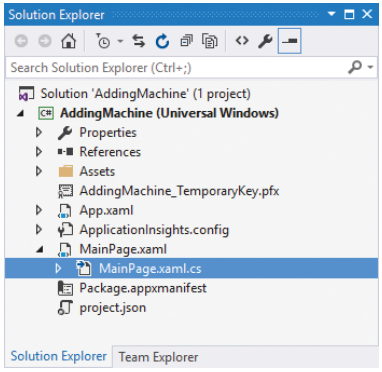


Figure 16-14 Locating the C# code-behind file.

If you actually take a look in the file named **MainPage.xaml.cs,** you will find that it seems not to contain much code:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
```

```
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at http://go.microsoft.com/
fwlink/?LinkId=402352&clcid=0x409

namespace AddingMachineTutor
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

Namespaces that contain the XAML objects.

Namespace thay contains our application.

Page is the parent class for the MainPage class.

The constructor for MainPage.

This method call sets up the page.

Most of the file consists of `using` statements that allow our program to make direct use of XAML classes without having to give the fully formed name for each of them. For example, rather than writing `Windows.UI.Xaml.Controls.Button`, we can write `Button` because the file contains the statement `using Windows.UI.Xaml.Controls`.

CODE ANALYSIS

Taking a look at the `MainPage` class

While the `MainPage` class looks a lot like other classes you have seen, some things about it are new.

Question: What does the namespace mean?

Answer: Namespaces let us give unique names to items in applications. You have seen that the C# `using` keyword allows you to tell the C# compiler to search a namespace for items that you use in your programs. The `namespace` statement in the program shows how a namespace is created. The effect of this statement is that the full name of the `MainPage` class is `AddingMachineTutor.MainPage`. Visual Studio uses the name of the application being created as an enclosing namespace for all the classes that are part of that application.

Question: Why is the class definition for `MainPage` so complicated?

Answer: When we made a class, it was very simple. We just wrote the `class` keyword followed by the name we had chosen for the class, as we did for the start of the `Contact` class declaration:

```
class Contact
```

If you take a look at the declaration of the `MainPage` class, you'll see lots of extra text:

```
public sealed partial class MainPage : Page
```

The first item worthy of note is `: Page` at the end of the declaration. This tells the C# compiler that the `MainPage` class is an extension of the `Page` class. You have seen this done before, when we made a set of sprite classes for Space Rockets in Space. We created different kinds of sprites by extending a parent type. Here we are creating a new kind of XAML page by extending the parent `Page` class. The `Page` class is provided as part of the set of resources used to create Universal Windows Applications.

Question: What does `sealed` mean?

Answer: You have seen that you can create new classes by extending parent ones. However, a class marked with `sealed` cannot be extended in this way. There is no reason for anyone to extend the `MainPage` class, so it is marked with `sealed` so that this cannot happen.

Question: What does `partial` mean?

Answer: You saw the keyword `partial` when you investigated how Snaps behaviors are created. The `partial` keyword tells the compiler that there may be other parts of this class in the application, stored in separate source files. In other words, this file holds part of the `MainPage` class. Partial classes make it easier to navigate large classes, which can be spread over several shorter files rather than stored in one large one. When the program is built, the contents of the C# code-behind file are combined with C# produced from the XAML page description to produce a C# object that represents the page on the screen.

The only method in the program is the constructor of the `MainPage` class. As you know, the constructor of a class is called when an instance of the class is created. All this constructor does is call the method `InitializeComponent`. If you took a look inside `InitializeComponent`, you would 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 not change the content of the method itself, as doing so would most

likely break your program. At this point we are deep in the engine room of XAML. I'm telling you about this so that you can understand that there is actually no magic here. 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 write statements in XAML to design and build our display.

Calculating the result

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

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

    float result = v1 + v2;

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

Method called to display the result.

Get the values that we are adding.

Calculate the result and then display it.

The `TextBox` display elements expose a property called `Text`, which can be read from or written to. Setting a value for the `Text` property will change the text in the `TextBox` on the screen. 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 to a numeric value if our program is to do any calculations on the value. You have seen the `Parse` method before. It takes a string and returns the number that the string describes. Each of the numeric types (`int`, `float`, `double`, etc.) has a `Parse` behavior that takes a string and returns 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 text boxes and then calculates a result by adding the values together.

Finally, the method takes the number that was calculated, converts it to a string, and then sets the text of the `resultTextBlock` to this string. `ToString` is the reverse of `Parse` (the "antiparse" 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 the code to run when the user presses the **Equals** button.

Events and programs

In Chapter 15, you saw how programs are made up of components that send messages to one another. When an alien collided with a rocket, the alien sent a message to the rocket saying, "You must now take damage." It did this by calling the `TakeDamage` method for the rocket. An event is a form of a message. When you talk to computer folk, opinions vary about the difference between an event and a message. My take is that an event is a kind of message that components in a program can subscribe to.

If this is confusing, think about the problem we are trying to solve. We want the program to display the result of the addition when a button on the screen is used. A button is a kind of display element that can generate events (in this case, "I have been clicked"), and our program would like to receive these events. Methods that run in response to events like these are called *event handlers*. From a C# point of view, there is nothing special about these methods. The important difference between an event-handler method and an "ordinary" one is that the event handler is called only when an event occurs.

It turns out that connecting a button to an event-handler method is actually quite easy, and Visual Studio will do most of the work for us. Start by returning to the XAML graphical editor in Visual Studio and selecting the button element (by clicking it with the mouse), as shown in **Figure 16-15**. With the button selected, you can move it around the screen or change its size, but what we want to do is edit the properties of the button and add an event handler for the `Click` event—in other words, specify a method that runs each time the button is clicked.

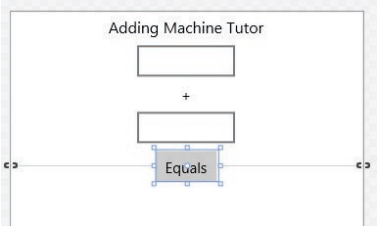


Figure 16-15 Selecting the button in the XAML editor.

Visual Studio provides a **Properties** pane where you can change the appearance and behavior of each object on the screen. This pane is usually in the bottom-right corner of the Visual Studio window. Selecting the button in the editor makes it the selected item in the **Properties** pane, too. (You can tell that the **Properties** pane is showing the correct properties because the value of the **Name** box displayed at the top is "equalsButton" and the Type is "Button.") **Figure 16-16** shows the properties of the `equalsButton` item.

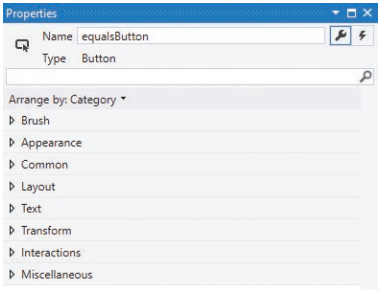


Figure 16-16 The button properties.

The properties are organized in categories. You can use them to change the appearance of the button, but we want to manage the events that the button can generate, so click the lightning bolt at the top right of the **Properties** pane to select the event panel, shown in **Figure 16-17**, which displays the events a button can generate.

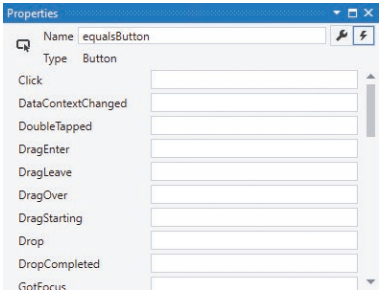


Figure 16-17 The list of button events.

If you wanted to, you could connect an event handler to every single one of these events, but we don't really have time for that just now. The event we want to connect to is at the top of the panel—the `Click` event. We can connect an event handler to this event by double-clicking in the empty text box next to Click. When you double-click, you'll see the method name appear, as shown in **Figure 16-18**.

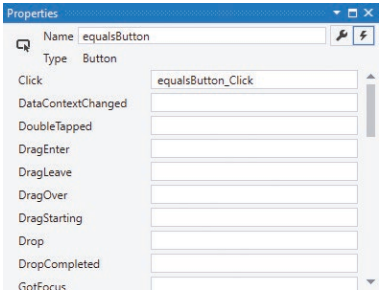


Figure 16-18 Click event handler added for the Equals button.

Two other things happen as well. First, Visual Studio adds a description of the event handler to the XAML for the **Equals** button:

```
<Button Content="Equals" Name="equalsButton" HorizontalAlignment="Center"
Margin="4"  Click="equalsButton_Click" ></Button>
```

Second, Visual Studio adds an empty event-handler method to the code behind the page:

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

When the program is compiled, the event described in the XAML is linked to the event-handler method so that when the button is pressed, the event-handler method runs. To make the calculator work, we just have to make the event handler call the method that will perform the calculation.

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

Demo 16-01 AddingMachineTutor
```

When the user clicks the **Equals** button, the `equalsButton_Click` method runs and calls the `displayResult` method to display the result.

CODE ANALYSIS

Event-handler methods

The event-handler method is provided for us automatically, but there are some things about it that you might want to consider.

Question: What are the parameters to the event-handler method?

> **Answer:** The event-handler method has two parameters. The first, named `sender`, is a reference to the object on the display that actually produces the event. In the case of the adding machine, this reference refers to the `equalsButton` object that the user clicked.

The second parameter, called e, provides details of the event that has occurred. In the case of a button click, there is not much information that can be added, but if events are generated by other actions—for example, pointer movement—the arguments parameter can provide details such as the position of the pointer when the event was produced.

Question: When does our program call the event-handler method?

Answer: The event-handler method is never called by our program. If the user clicks the button, the method is called, but our program will never actually call this method itself. This is quite a different model from the one what you are used to. Most of the time, our adding machine program will be doing nothing; it will just be waiting for the user to perform an action.

Question: What happens if the event-handler method takes a long time to complete?

Answer: It is very important that an event handler completes as quickly as possible. As long as the event handler is running, the rest of the user interface is unable to do anything. We have all experienced the awful feeling of loss of control when an application becomes unresponsive. The cause of this unresponsiveness is always one of the event-handler methods in the application getting stuck.

Dealing with errors

The solution that we have come up with does work, and it will provide a working adding machine. However, it is not a very user-friendly program. For example, the user might type text into the numbers text boxes as shown in **Figure 16-19**.

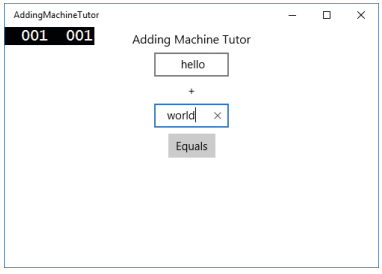


Figure 16-19 Invalid numbers have been entered here.

We already know that this will cause problems with the `Parse` method, which will promptly throw an exception and stop the program. We also know that we can catch exceptions and deal with them (you saw this in Chapter 11). In the case of the adding machine program, the best way to deal with these kinds of errors is to display a message:

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

        float result = v1 + v2;
        resultTextBlock.Text = result.ToString();
    }
    catch
    {
        resultTextBlock.Text = "Invalid number";
    }
}

Demo 16-02 AddingMachineTutorErrors
```

If either of the calls to the `Parse` method in the `try` block throws an exception, the program transfers execution to the `catch` block, which displays the message "Invalid number" in `resultTextBlock,` as shown in **Figure 16-20**.

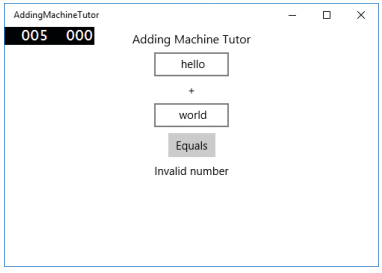


Figure 16-20 Invalid numbers error display.

Using TextBox properties to improve the user interface

The error handling for our adding machine tutor is not perfect. While the program catches errors, it doesn't actually tell the user which of the values is wrong when it runs. We can improve on this user interface by using some more properties of the `TextBox`. There are lots and lots of things that we could do. What we are going to do

is change the background of a text box with an invalid entry so that it is red. The text box will appear as shown in **Figure 16-21**.

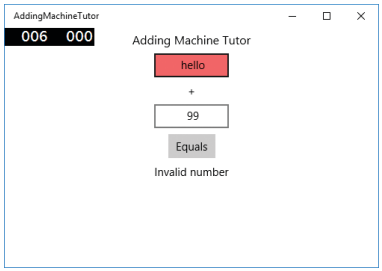


Figure 16-21 Highlighting invalid entries by using a property.

If an entry is invalid, it will be colored red. In the XAML universe, elements that are drawn on the screen are drawn with a brush object. There are a number of different brush classes available; in fact, the XAML `Brush` is a nice example of class-based design. `Brush` is the parent class of a number of different kinds of brushes that can draw patterns or images. The brush we want to use is the `SolidColorBrush`. We can use it to create a brush with a particular color:

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

This creates a `Brush` reference named `errorBrush`, which is set to a solid red brush. We can now set the background of a `TextBox` to be drawn with this brush:

```
firstNumberTextBox.Background = errorBrush;
```

This statement sets the background of `firstNumberTextBox` to red to indicate that an error has been detected. This might seem like a long way around setting a color value, but it actually gives you a lot of flexibility. You could create an image that represents an error and then use it as the background of the `TextBox` instead.

We can set the background color of the text box when the `Parse` method throws an exception, but there is also a way to convert text to a number that doesn't use exceptions to indicate that an error has occurred. The `TryParse` method tries to parse the string and returns `false` if the attempt fails. You can see how to use it in the following statements:

Adding "out" means the method must put a value into this argument.

```
float v1;
if (float.TryParse(firstNumberTextBox.Text, out v1) == false)
```

```
{
    firstNumberTextBox.Background = errorBrush;
}
```

If TryParse fails, the background of the TextBlock is set to red.

This code will set the background of the `TextBox` to red if the text entered is not a valid number.

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

private void displayResult()
{

    float v1;
    float v2;
    bool validValues = true;

    if (float.TryParse(firstNumberTextBox.Text, out v1)==false)
    {
        validValues = false;
        firstNumberTextBox.Background = errorBrush;
    }

    if (float.TryParse(secondNumberTextBox.Text, out v2)==false)
    {
        validValues = false;
        secondNumberTextBox.Background = errorBrush;
    }

    if (validValues)
    {
        float result = v1 + v2;
        resultTextBlock.Text = result.ToString();
    }
    else
    {
        resultTextBlock.Text = "Invalid number";
    }
}

Demo 16-03 AddingMachineTutorFaultyErrorDisplay
```

This flag indicates whether all values are valid.

Try to parse the first value.

If the parse fails, set the flag to indicate this input is invalid.

Repeat for the second value.

CODE ANALYSIS

Faults in displayResult

This dislayResult method looks like it should work, but unfortunately it has a serious bug.

Question: What's wrong with displayResult?

Answer: The problem with displayResult is not apparent the first time you use it. If you test the program just once, you find that if you enter valid information, the program displays the correct result. If you run the program again and enter invalid values, the appropriate TextBox turns red to indicate an error. However, if you enter some valid values after you have tried some invalid ones, the background stays red. This is not particularly surprising, as there is nothing in the program that resets the backgrounds for the text boxes once they have been turned red to indicate that a value is invalid.

Question: How do we fix the background colors?

Answer: We can fix this error by restoring the background color to the usual one if a value in a TextBox is found to be okay. We do have to be careful here, though, because we can't just assume that everyone uses the color white as the background for their text. Some people might be using fancy color schemes on their PCs. The good news is that we can read and store the original background color of a TextBox and just put the original background brush back when the value is found to be valid. The program can sample the original TextBox background color in the constructor for the page and then use this to set the background color of correct entries.

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

Brush correctBrush;                                    Brush used to indicate "correct."

public MainPage()
{
    this.InitializeComponent();

    correctBrush = firstNumberTextBox.Background;      Copy original background color
}                                                      into the correct brush.

private void displayResult()
{

    float v1;
    float v2;
```

```
    bool validValues = true;

    if (float.TryParse(firstNumberTextBox.Text, out v1) == false)
    {
        validValues = false;
        firstNumberTextBox.Background = errorBrush;
    }
    else
        firstNumberTextBox.Background = correctBrush;

    if (float.TryParse(secondNumberTextBox.Text, out v2) == false)
    {
        validValues = false;
        secondNumberTextBox.Background = errorBrush;
    }
    else
        secondNumberTextBox.Background = correctBrush;

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

This code runs if the value is valid.

Set the background color to indicate the value is correct.

Demo 16-04 AddingMachineTutorFixedErrorDisplay

MAKE SOMETHING HAPPEN

Make some different tutors

You can use the pattern for the adding machine to create programs to do subtraction, multiplication, and division. You could even make one "monster tutor" that had different areas of the screen for doing each of these calculations, or make a version that took in two numbers and displayed their sum, product, and difference.

What you have learned

In this chapter you moved away from the Snaps environment and created your first Windows 10 Universal Application. You saw how to use XAML to describe the arrangement and properties of the elements on the application page and how to use Visual Studio to edit and preview your XAML designs. You discovered that elements described in the XAML file are revealed as objects inside the C# program file that sits behind the user interface. Programs can update display elements by writing to properties in the elements, and read information from the display by reading from the elements. We identified three XAML elements to start with: a `TextBlock` that displays text, a `TextBox` into which users can enter raw text, and a `Button` element that can be used to trigger events.

The `Button` element has events that can be used to run C# methods in response to user actions. Unlike in previous programs, where our code starts running when the program begins, a XAML application has C# code that is bound to events generated by elements in the display. The XAML that describes a button can contain a `Click` property that identifies the C# method that will run when the button is clicked. You have seen how programs can be created to work in this way and how outputs can be generated in response to user actions.

Finally, we investigated some simple error handling and took our first steps toward creating properly user-friendly interfaces.

Here are some questions you might want to ponder about XAML-based user interfaces.

How different are our programs from professional ones?

The programs that we are now writing use all the same techniques and display elements as full-size professional ones. As I said much earlier in this book, the only difference between the programs we are writing and professional programs is that we are not selling our programs yet. The use of XAML to design the program and the way that methods in the code respond to the events generated by XAML display elements are exactly the same in our programs as they are in larger ones.

Can I create my own elements to place on the display?

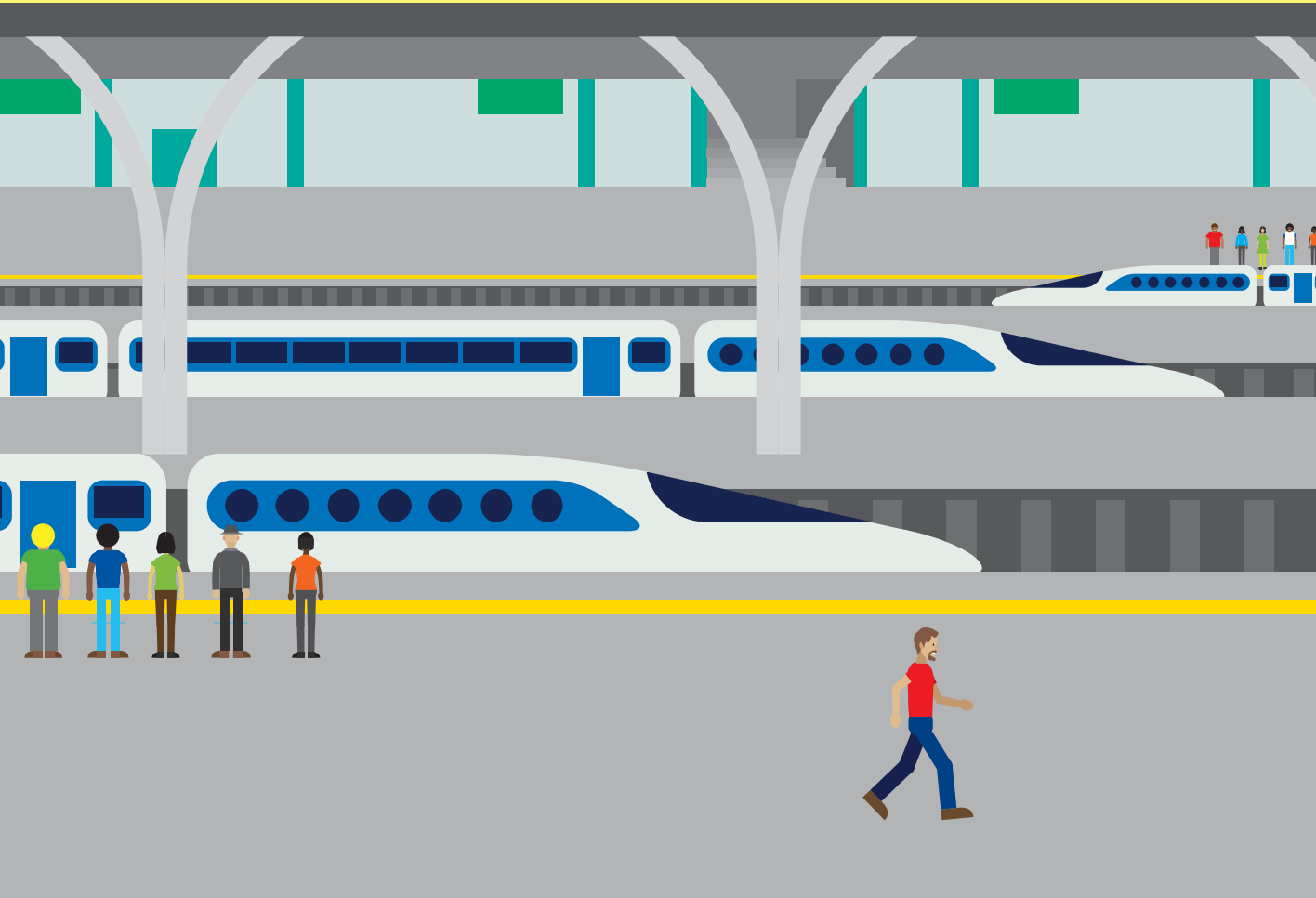
Yes. Learning how to do this is beyond the scope of this book, but the fact that all the elements on the screen have been built from a class hierarchy means that you can extend them to add your own behaviors. You can also create your own custom controls that contain a number of control elements that can then be manipulated by Visual Studio.

How can I make really good-looking user interfaces?

The XAML that we have created so far is very utilitarian. It does the job, but it is not very pretty. Fortunately, there is a tool called Blend that is supplied as part of a Visual Studio installation. Blend provides a designer-focused view of the user interface design. You can use Blend to create graphical effects and animations that are applied to your components and create and use display templates that can be applied to elements in your user-interface designs. It's important to remember that, at the end of the day, the output will still be a text file that contains XAML descriptions of the display elements. The XAML language gives an incredible level of control over how you display elements, and Blend provides a great place to do the design work. You can explore Blend (but it is a very complex program) by right-clicking any of the XAML files in Solution Explorer and selecting **Design in Blend**.

17

Applications and objects



What you will learn

In the last chapter, we created a very simple adding machine that showed how applications can communicate with their users. You saw how to use objects to represent elements in a user interface and how a program can change the properties of the objects to update information that is displayed to users.

In this chapter, you are going to examine how user interfaces and objects can be made to interact in a well-structured application You'll also have some fun adding pictures and sounds to a Universal Application and find out how to use the `ComboBox` element to allow a user to make a choice.

Making a calculation quizzer

The adding machine we made in Chapter 16 is a good program if you want to perform calculations and view the results, but it is not a very good tool for practicing how to do addition. A better practice program would be one that displays a question, asks for an answer, and then indicates whether the answer is correct.

As a starting point, we could use an application design such as you see in **Figure 17-1**. The question appears at the top, there's a text box where the user types in an answer, and the **Check Answer** button is used to verify the answer (when the button is pressed, the program indicates whether the answer is correct). The user can press **Next Question** to move to another question.

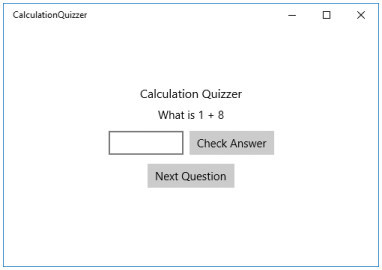


Figure 17-1 The Calculation Quizzer.

This program looks similar to the adding machine program that we got working in the last chapter, so it seems like it should be easy to write. We just have to change the way the program works behind the buttons on the page.

Objects and user displays

In the Time Tracker application, we used a `Contact` object to hold information about each contact. For example, a `Contact` object holds the name of the contact. But the `Contact` class doesn't contain any code that actually displays that object on a screen. Programs are frequently structured to make a strong distinction between objects that manage business information (such as a contact's name) and objects that perform input and output. We call objects like the `Contact` object *business objects* because their job is to store business information, not interact with the user.

For the Calculation Quizzer, we can regard the "quizzing" behavior as a form of business object and separate it from the part of the program that drives the display. The display asks the quiz object, "Give me a question." Then, when the user enters an answer, the display asks the quiz object, "Is this answer correct?" There are two huge benefits in making the quiz application work this way: it becomes easier to test, and it's much more flexible.

Testability

If we make quiz behaviors part of the display page, we can't test them automatically. I like programs that can test themselves, not ones where I have to sit at the computer, type things in, press buttons, and see what comes back. In the case of our simple quiz program, there is not a great deal to test, but if we were working on a more complicated application—for a bank, for example—we would not want to perform thousands of transactions by hand to see whether the program works correctly. We would want to check the bank balance, pay money in, and then check again to see whether the balance changed correctly, without having do this via the screen. Testing this functionality would be much easier if the bank account transactions were performed by an object external to the display page, because then we could write some code that would perform lots of transactions and check the results for us.

Similarly, if we know what the addition quiz object is supposed to do, we can ask it for a question, evaluate the answer ourselves, and then see whether the quiz object thinks the answer is correct. All this can be performed automatically.

Flexibility

After people work with our program for a while, they'll see that it makes practicing addition very easy. You might find that they now want a program to test their knowledge of subtraction and also multiplication. And then they'll want to expand the program again to cover subjects such as history. If the testing behavior is built into the display, we have to make a new display for each type of quiz. But if the quiz and the display are separate objects, we can swap one quiz object for a different one, and the user-interface design can remain the same. If we make good use of C# interfaces, the mechanism you saw in Chapter 15, we can create lots of new quiz types and just plug them into our application. We could even create a general-knowledge quiz that used questions from all our different quiz objects.

Creating a quiz object

When the quiz program starts, the display page will create a quiz object and then use this object to display questions and answers to the user. We can think of the relationship between the quiz object and the display page in terms of the messages that are sent when the quiz takes place. There are three things that the display page needs:

- It needs to get the text to display in the question part of the page.
- It needs to check the answer that the user has entered.
- It needs to move on to the next question.

These actions correspond broadly to the buttons on the page, as shown in **Figure 17-2**.

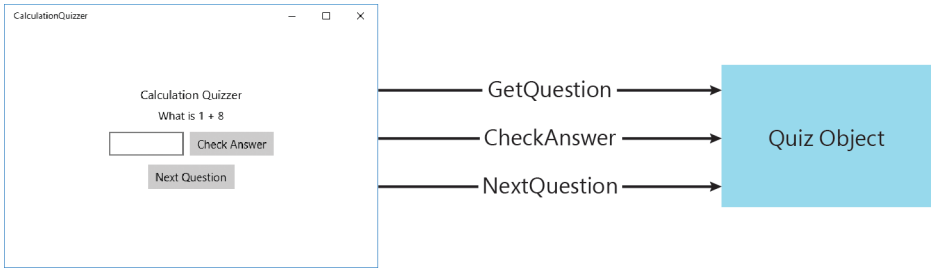


Figure 17-2 A quiz object and its relationship to the display page.

We can express these actions in a C# interface:

```
/// <summary>
/// An object that can be used to generate
/// and test quiz questions
/// </summary>
interface IQuizObject
{
    /// <summary>
    /// Gets the text for a question
    /// </summary>
    /// <returns>the question text</returns>
    string GetQuestion();

    /// <summary>
    /// Checks to see if an answer is correct
    /// </summary>
    /// <param name="answer">answer to be tested</param>
    /// <returns>true if the answer is correct</returns>
    bool CheckAnswer(string answer);

    /// <summary>
    /// Moves onto the next question
    /// </summary>
    void NextQuestion();
}
```

You have seen interfaces before. A class can implement this interface and then be regarded in terms of this ability. In other words, we can take any class we like, give it

the three methods identified in the interface, and it can then be referred to by a reference of type `IQuizObject`.

Professional comments

Earlier in this book, we considered what makes a program "professional." I think one thing that distinguishes a professional program is the level and quality of the comments in the source code. You can see that the `IQuizObject` interface actually has more comments than C# statements. The comments themselves are in the XML-like format introduced in the section "Adding IntelliSense comments to your methods," back in Chapter 8. Recall that the comments are designed to be read by Visual Studio and used to provide IntelliSense information that pops up when the program is being edited.

In **Figure 17-3,** I'm using the Visual Studio code editor to create a new class, named `FakeWrongQuiz`, that implements the `IQuizObject` interface. When I rest the cursor over the name of the interface, I get a pop-up window that includes the information about the interface that was added in the comment.

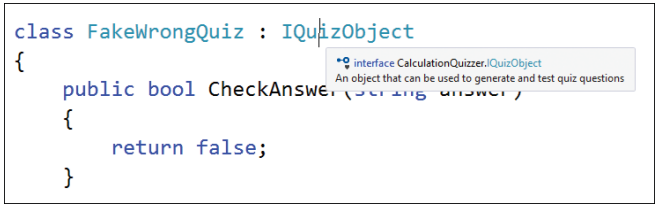


Figure 17-3 IntelliSense help is part of making a program professional.

Having information like this available to a programmer makes it much easier to create and use objects developed by other people. Remember that Visual Studio even creates the comment templates for you; all you have to do is type three slashes (***///***) into the Visual Studio editor above the method or class that you want to document. When you add comments to methods, you can provide a summary of the method, a description of each parameter, and a description of what the method returns. When I see a program that contains comments like these I start to consider it a "professional" piece of work.

Making a fake object

The class `FakeWrongQuiz`, shown in the following code, provides implementations of all the methods defined in `IQuizObject`, but they don't do anything much. The question is always the same string, and the answer is always wrong.

```
class FakeWrongQuiz : IQuizObject
{
    public bool CheckAnswer(string answer)
    {
        return false;
    }

    public string GetQuestion()
    {
        return "The answer to this question is always wrong";
    }

    public void NextQuestion()
    {
    }
}
```

Implements IQuizObject

Always returns false.

Always returns the same question.

No next question, so this method does nothing.

This class doesn't look very useful, but actually it is. A programmer can use this class to test the user interface that she is developing. She can have this fake quiz ready well before the real quiz objects are finished, which means that the display page and the quizzes can be developed at the same time. Perhaps a team of friends could create the quiz objects and you could build the user interface. When all of you have finished writing your code, you can plug your classes together and have a working program. **Figure 17-4** shows the "fake wrong" quiz being used.

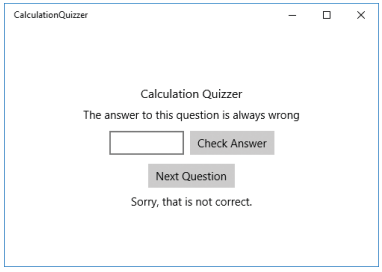


Figure 17-4 A fake object can be used for testing.

We could also create a "fake right" quiz object that could be used to test the behavior of the program when the user gets the answer right.

Creating an addition quiz object

The addition quiz object picks two random numbers, which it then uses to generate a question. The random numbers are produced by a random-number generator, which is a member of the object.

```
/// <summary>
/// A quiz object that implements an addition quiz
/// </summary>
class AdditionQuizObject : IQuizObject
{
    /// <summary>
    /// Random number generator used by the quiz
    /// </summary>
    private Random rand = new Random();

    /// <summary>
    /// Current question being asked by the object, as a string
    /// </summary>
    private string currentQuestion;

    /// <summary>
    /// Answer value, which is an integer
    /// </summary>
    private int currentAnswer;

    public string GetQuestion()
    {
        //Just return the current question
        return currentQuestion;
    }

    public bool CheckAnswer(string answer)
    {

        int answerValue;

        // Convert the parameter into a number
        if (int.TryParse(answer, out answerValue))
        {
            // If the number conversion succeeds
            // check against the answer
            if (answerValue == currentAnswer)
                // return true if the answer is correct
```

```
                return true;
            }
            // Either the answer was wrong or the user did
            // not enter a number. Return false
            return false;
        }

        public void NextQuestion()
        {
            // Generate two numbers in the range 0 to 9
            int firstNum = rand.Next(0, 10);
            int secondNum = rand.Next(0, 10);

            // Store the question string
            currentQuestion = "What is " + firstNum + " + " + secondNum;

            // Store the correct answer
            currentAnswer = firstNum + secondNum;
        }

        public AdditionQuizObject()
        {
            // When the object is created, set up
            // the first question
            NextQuestion();
        }
    }
```

CODE ANALYSIS

Taking a look at `AdditionQuizObject`

Question: Why does the `CheckAnswer` method check a string rather than a number?

Answer: The `CheckAnswer` method in the quiz object is given answers to compare with the correct one. If the answer that is supplied does not match the correct one, the method returns `false` to indicate that the answer is wrong. This object is implementing a mathematical quiz, which works with numbers, but the `CheckAnswer` method checks a string rather than a number. However, this is not a mistake. It means that we could make a quiz that accepted text as answers—for example, the surname of the first president of the United States—as well as numbers. It makes for slightly more work for the addition quiz object because the quiz object must convert the text that is being checked into a number, but it makes the quiz much more flexible.

Question: Is there anything to stop a cunning programmer from peeking at the results in the quiz?

> **Answer:** Yes, there is. The question text and the all-important answer have been designed as private data members of the `AdditionQuizObject` class. Only methods running inside `AdditionQuizObject` have access to them, which means that there is no way that a sneaky programmer could read the answers out of the quiz object. This is one of those situations where protecting the data in an object is a very good idea.

Question: How can we make the addition problems more difficult?

> **Answer:** One way would be to extend the range of numbers that the program produces. This is controlled by the range of the random numbers that are used by `NextQuestion`. At the moment, the method uses values in the range 0 to 9, but this could be extended (and even start with a negative number) if you wanted to make the problems harder.

Creating the quiz display page

The quiz display page is described in XAML. We can use the `StackPanel` container to stack up the elements on the screen. We can also use a horizontal `StackPanel` so that we can put the answer the user enters and the **Check Answer** button on the same line on the screen.

```
<StackPanel VerticalAlignment="Center">
    <TextBlock Text="Calculation Quizzer" TextAlignment="Center" Margin="4"
     FontSize="16"></TextBlock>
    <TextBlock Name="questionTextBlock" Text="" TextAlignment="Center" Margin="4">
     </TextBlock>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center" Margin="4">
       <TextBox Name="answerTextBox" Width="100" Margin="4"
        TextAlignment="Center"></TextBox>
       <Button Content="Check Answer" Name="checkAnswerButton"
        HorizontalAlignment="Center Margin="4"
        Click="checkAnswerButton_Click" ></Button>
    </StackPanel>
    <Button Content="Next Question" Name="getNextQuestionButton"
     HorizontalAlignment="Center"Margin="4"
     Click="getNextQuestionButton_Click" ></Button>
    <TextBlock Name="resultTextBlock" Text="" TextAlignment="Center" Margin="4">
     </TextBlock>
</StackPanel>
```

The code behind the XAML page sets up the quiz in the constructor for the class and then provides event handlers for the two buttons on the screen:

```
public sealed partial class MainPage : Page
{
    IQuizObject activeQuiz;

    public MainPage()
    {
        this.InitializeComponent();

        activeQuiz = new FakeWrongQuiz();
        questionTextBlock.Text = activeQuiz.GetQuestion();
    }

    private void checkAnswerButton_Click(object sender, RoutedEventArgs e)
    {
        if (activeQuiz.CheckAnswer(answerTextBox.Text))
        {
            resultTextBlock.Text = "Correct! Well done.";
        }
        else
        {
            resultTextBlock.Text = "Sorry, that is not correct.";
        }
    }

    private void getNextQuestionButton_Click(object sender, RoutedEventArgs e)
    {
        activeQuiz.NextQuestion();
        questionTextBlock.Text = activeQuiz.GetQuestion();
        answerTextBox.Text = "";
        resultTextBlock.Text = "";
    }
}
```

Quiz that the display page is using.

Constructor

Set up the active quiz.

Put the question into the text block.

Runs when the answer is being checked.

Check whether correct answer entered.

Advance to the next question.

Set text for the next question.

Demo 17-01 Simple Quiz

CODE ANALYSIS

The complete program

The code here is quite compact, but it's still interesting.

Question: Which particular quiz is this program using?

Answer: The variable `activeQuiz` is set to the currently active quiz. When this version of the quiz is started, it is set to an instance of `FakeWrongQuiz`.

Question: How do we make this into an addition quiz?

Answer: We just have to set `activeQuiz` to refer to an instance of `AdditionQuiz`.

Adding sound and pictures

The quiz program works very well, but the display is a bit boring. You could add some sounds to start with and maybe some pictures, too.

Adding sound

Let's start with sound. We've used Snaps to make sounds in the past, but now you are going to find out how to add sound effects to a Windows 10 Universal Application. The range of XAML display elements is not restricted to ones that can display images; we also have elements that can make sounds and even play videos.

Here is the XAML that creates a media element with the name `soundMediaElement`. We can use this to play sounds in our programs. We can put it anywhere on the screen because it is just providing sound output. You can also use the `MediaElement` type to play video, but we don't need to for this program.

```
<MediaElement Name="soundMediaElement"></MediaElement>
```

We add sounds to Windows 10 applications in the same way as we have added sounds to Snaps programs in the past. We use .WAV sound files and drag them into our program's assets, as shown in **Figure 17-5**.

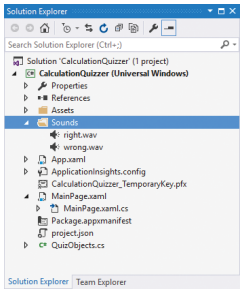


Figure 17-5 Adding sound items to the Calculation Quizzer.

Now that we have the sound item and the XAML element to actually make the sound, we have to create the C# code that will play the sound.

```
private void checkAnswerButton_Click(object sender, RoutedEventArgs e)
{
    if (activeQuiz.CheckAnswer(answerTextBox.Text))
    {
        resultTextBlock.Text = "Correct! Well done.";
        Uri soundsouce = new Uri("ms-appx:///Sounds/right.wav");
        soundMediaElement.Source = soundsouce;
        soundMediaElement.Play();
    }
    else
    {
        resultTextBlock.Text = "Sorry, that is not correct.";
        Uri soundsouce = new Uri("ms-appx:///Sounds/wrong.wav");
        soundMediaElement.Source = soundsouce;
        soundMediaElement.Play();
    }
}

Demo 17-02 Simple Quiz with Sound
```

Create the Uri, set the source, and play the media element.

Use a different sound for wrong answers.

The key to understanding how the sound works is to understand the use of `Uri` (or uniform resource indicator). This is an element that refers to a particular resource. A program can use lots of kinds of resources, including sounds, images, and files. The resources can be local to the computer, held inside the application itself, or available via a network connection. The uniform resource indicator is a way to create references to these resources that can be used by the XAML elements. The `Uri` can be created from a string that specifies the location of the resource. Adding the prefix "ms-appx:///" to a resource's location means that the resource is content that is stored within the application.

In **Figure 17-5,** you can see that I've put the sounds in a folder named **Sounds** in the project, so the address of my sounds in the program must include this folder in the path to the resource.

Once I have the `Uri` that provides the address of the resource, I can then set the `Source` property of the `soundMediaElement` to this address. (Some XAML elements have a `Source` property that specifies where they are going to get their content from.) The `Source` is set to the `Uri` of the item that is to be played by the `MediaElement`. The final statement, the `Play` method call, makes a `MediaElement` play the media item that is assigned to its source. In this case, the appropriate sound effect is played. You can change the sound that is played by just changing the source element.

Adding images

You can add images to XAML pages by using the `Image` element. You could just add a picture to the user interface, but what I'd like to do is add a background behind the entire application so that the page appears as shown in **Figure 17-6**.

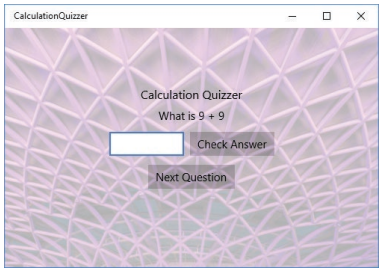


Figure 17-6 Adding a background to the page.

The background should fill the entire screen. I can achieve this by exploiting the way that the `Grid` display element works in XAML. We can use the `Grid` display element to lay out a screen of items. It is slightly harder to use than the `StackPanel` control, but it gives you a lot of control over what portion of the display is allocated to each element.

When you create a XAML project, the display is made up of a grid containing just one cell, which forms the entire screen. All the items in this grid are drawn on top of one another in the order that they are given in the `Grid` element. So if I make an `Image` item and expand it to fill the entire screen, it will provide a rather nice background. I can actually do all this from within the XAML design. Here is how a background image is added to the page:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Image Source="ms-appx:///Images/kingscross.jpg"
    Opacity="0.3"
    Stretch="Fill">
  </Image>
  <StackPanel VerticalAlignment="Center">
  </StackPanel>
</Grid>

Demo 17-03 Simple Quiz with Background
```

StackPanel that makes up the application display.

The `Opacity` value, which is given in the range 0 to 1, is useful because it controls how much of the background shows through the image. If I just put the image behind the controls, they become hard to see against the picture. The smaller the value, the fainter the image is. An `Opacity` setting of 1 is a solid image, but a value of 0 would

make the image completely transparent. From my tests I found that a value of 0.3 works well. The image is visible, but you can still see the controls.

MAKE SOMETHING HAPPEN

Create an impressive quiz

You can use images and sounds to make a very interesting quiz display. You can change the source of an `Image` from within the program as well as from the XAML, so you could make the background of the program change if the user gets the answer wrong. You can also lay images on top of each other and use their opacity settings to merge them together. Try making a really impressive version of the quiz program, or use these techniques to improve other programs that you have written.

Supporting multiple quizzes

Just as we expected, people now want a version of the program that supports different types of quizzes. We can start by building a quiz to practice three more arithmetic activities: subtraction, multiplication, and division.

Creating quiz classes

You might think it would be a good idea to extend the `AdditionQuizObject` to produce new classes that can perform these functions, but I don't think this is a very good plan. You should extend a class and then override the methods in the parent class to produce a less-abstract version of the child, not a completely different one.

For example, in a bank application, you would extend the `Account` class to produce `CheckingAccount`, and then extend `CheckingAccount` to produce `CheckingAccountWithOverdraft`. These classes all do the same fundamental thing; it is just that the child classes serve slightly different scenarios. However, extending an addition object to make a subtraction object seems wrong to me because the child is doing something completely different from the parent, not providing a more specialized version of it. What we really need to do is be more abstract and create a `CalculationQuizObject` class, which is then extended to produce the different kinds of calculation quizzes. **Figure 17-7** shows a design for a class hierarchy of quizzes.

PROGRAMMER'S POINT

You might not work out the best design at the start

As far as C# is concerned, the compiler doesn't care about the meaning of your classes and whether or not a design is perfect. And neither will users of your program. As long as they get a quiz that works, they will be happy. There is a strong argument for having a mindset along the lines of "As soon as you come up with a design that you know will work, you can stop designing the program and start building it. At least that way the customer will get something."

My experience has been that I can spend a lot of time searching for the perfect design for a solution, and then when I start to make the program, I realize a much better way of doing it. This is true in many fields, where prototypes are used to test out ideas and inform the production process. Programmers have a technique called "refactoring," which is where they change the arrangement of the classes in their programs during development. The more advanced versions of Visual Studio provide powerful tools that can help programmers do this.

There is nothing wrong with improving the design of your software as you build it, but for me the most important thing is that you should try very hard to make sure that your proposed design will actually work before you start to build it. We've already discussed how painful being blindsided can be; it is best to make sure that it doesn't happen to you.

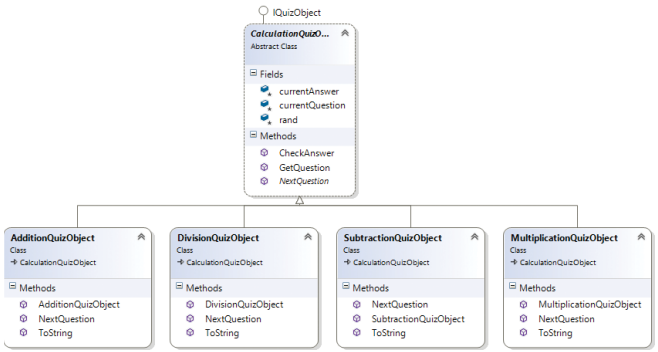


Figure 17-7 Quiz class hierarchy.

The only method that is different in each of the child classes is the `NextQuestion` method, which sets up the question text and calculates the answer. Here is the `NextQuestion` method from the `MultiplicationQuizObject` class. It overrides an abstract method in the `CalculationQuizObject` class to provide the behavior for multiplication questions.

```
public override void NextQuestion()
{
    int firstNum = rand.Next(0, 10);
    int secondNum = rand.Next(0, 10);
    currentQuestion = "What is " + firstNum + " * " + secondNum;
    currentAnswer = firstNum * secondNum;
}
```

The class diagram in **Figure 17-7** was actually produced by Visual Studio 2015 from the program that I created. **Figure 17-8** shows how you do this: right-click any of your source files in Solution Explorer, select **View Class Diagram**, and Visual Studio draws a diagram that shows how the classes are related. You can use such a diagram to create new classes in your solution.

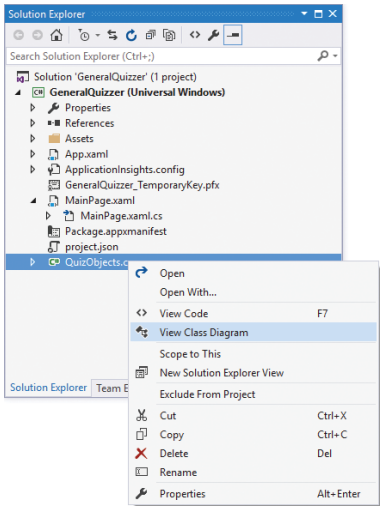


Figure 17-8 Use these options to view a class diagram.

Selecting quiz types with a ComboBox

Now that we have a set of calculation quiz classes, we need a way for users to select the type of questions that they want to answer. We can use a XAML control called a `ComboBox` to allow the user to select the quiz topic. **Figure 17-9** shows how this looks when the program runs.

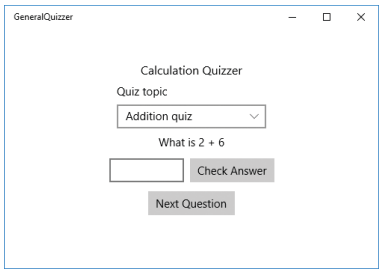


Figure 17-9 Quiz selector combo box, with the addition quiz selected.

If the user wants to change to a different quiz, she can open the combo box and pick a new quiz type. **Figure 17-10** shows the selection options when the combo box is opened.

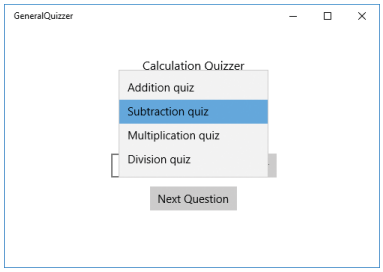


Figure 17-10 Users select a quiz type using the combo box.

Users will like the idea of this selection option because they will have seen it on lots of other programs. Now we have to add the combo box to our program. Here is the XAML that describes the `ComboBox`. We refer to the `ComboBox` in our program by the name `quizTopicComboBox`.

```
<ComboBox Header="Quiz topic"
    Name="quizTopicComboBox"
    Width="200" Margin="4"
    HorizontalAlignment="Center"
    SelectionChanged="quizTopicComboBox_SelectionChanged">
</ComboBox>
```

Give the combo box a quiz topic.

Event that fires when the selection changes.

Now we have to set up the `ComboBox` with the options that we want users to be able to select.

```
public MainPage()
{
    this.InitializeComponent();

    quizTopicComboBox.Items.Add(new AdditionQuizObject());
    quizTopicComboBox.Items.Add(new SubtractionQuizObject());
    quizTopicComboBox.Items.Add(new MultiplicationQuizObject());
    quizTopicComboBox.Items.Add(new DivisionQuizObject());

    quizTopicComboBox.SelectedIndex = 0;
}
```

Add quiz objects to combo box.

Select the initial quiz object.

A `ComboBox` holds a collection of `Items` and lets the user select one item from that collection. A program can add items to the collection for the user to choose from. In the case of the **GeneralQuizzer** application, the items are the different quiz objects. We need to add each type of quiz to the items in `quizTopicComboBox` when the program starts. The best place to do this is in the constructor of the page. The `Items` in the `ComboBox` work exactly like the `List` collections you have seen before. A program can add items to the list, and they will be managed by the `ComboBox`.

Once we have set up the items in the `quizTopicComboBox`, we have to make sure that one of the items is selected when the program starts. A program can do this by setting a value for the `SelectedIndex` property of the `quizTopicComboBox`. In the code above, I've set the `SelectedIndex` value for the `quizTopicComboBox` to the first element (the one with the index of 0). The effect of this action is the same as though the user had opened the `quizTopicComboBox` on the screen and then selected the item at the top of the list of options. It also causes the `SelectionChanged` event to be raised in `quizTopicComboBox`.

```
IQuizObject activeQuiz;

private void quizTopicComboBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    activeQuiz = (IQuizObject) quizTopicComboBox.SelectedItem;
    setupNextQuestion();
}
```

The currently active quiz.

Get selected item; use it to set the value of ActiveQuiz.

Get the next question from this quiz.

CODE ANALYSIS

Selection-changed events

We've not worked with a `ComboBox` element before. It presents some interesting questions we need to consider.

Question: When does the `SelectionChanged` method run?

> **Answer:** The answer to this is both simple and complicated. The method runs when the user changes the selection in the `ComboBox` as the program is running. However, the method also runs if anything in our program changes the selection in the `ComboBox`. We use this mechanism to good effect when the program starts and we need to select an initial quiz. One thing that always ends badly (and I have done this) is when the `SelectionChanged` event hander changes the selected item in the `ComboBox`, causing another `SelectionChanged` event. This ends up producing an infinite sequence of changes that has been known to lock up my computer.

Question: Where do the names of the items come from?

> **Answer:** In the screenshots of the program, it looks like the `ComboBox` shows the name of each kind of quiz and lets the user select the one that he wants. But where does this name come from? The answer is that the `ComboBox` uses the `ToString` behavior to get the text that represents each item that is being selected. You have seen `ToString` before; it is how we ask any object to give us a string that describes its contents. For example, inside the `MultiplicationQuizObject`, we have the following `ToString` method.

```
public override string ToString()
{
    return "Multiplication quiz";
}
```

Question: How is the item from the `ComboBox` used to set the active quiz?

> **Answer:** The `ComboBox` class can be used to select any kind of item. It does this by storing a list of objects that the user can choose from. In the case of the `quizTopicComboBox`, the items in the list are all objects that implement the `IQuizObject` interface (that is, we can use them for quizzes). However, the list of items in the `ComboBox` must be a list of objects. This means that the program must convert the selected item (which is a reference to an object) into a reference to an `IQuizObject` that can be used in the quiz. It uses a cast to perform this conversion. We have used casts before when we want to tell the compiler that it is okay to convert from one type to another. The type we want to use is given in parentheses in front of the value that we want to convert.

```
activeQuiz = (IQuizObject) quizTopicComboBox.SelectedItem;
```

MAKE SOMETHING HAPPEN

Make a timed general-knowledge quiz program

You can now make a complete quiz program. You could create a set of different quiz objects that handle different subjects and then allow the user to select them. You could even allow the user a certain number of seconds to answer the question. A quiz program could record the time when the question was displayed and then check the time when the answer button is pressed. The faster the question is answered, the higher the score. It is possible to subtract one `DateTime` value from another and generate a `TimeSpan` value that a program can use to determine the size of a time interval.

What you have learned

In this chapter, you have learned the proper way in which a user interface should be created for an object-oriented solution. You have seen that the user interface should be a very thin layer of code that delivers messages to an object that provides the behaviors of the program. Working this way brings many advantages. It is much easier to test the object that implements the application because you can write programs that can simulate the actions of the user interface and check the responses of the object that makes it work. You can also create "fake" versions of either the user interface or the business objects that can be used to test the system as it is developed. This also makes it possible to work on both user interface and business behaviors at the same time. We have seen that the C# interface mechanism is very useful in this situation because it can be used to set out the nature of the method calls that will be used to pass the messages.

You also found another use for a class hierarchy when we created a collection of different numeric quiz objects that differed only in the generation of the actual question itself and were able to share all the other behaviors. From a user-interface perspective, you learned how to add images and sounds via the `MediaElement` and `Image` elements. Finally, you saw how a user can pick from a range of options by using a combo box.

Question: Why is the word interface used in such a confusing way?

I have a real problem with this one. Whenever we talk about interfaces, we have to be careful about what we actually mean. The user interface is the collection of elements that makes up the experience users have when they interact with a program. A C#

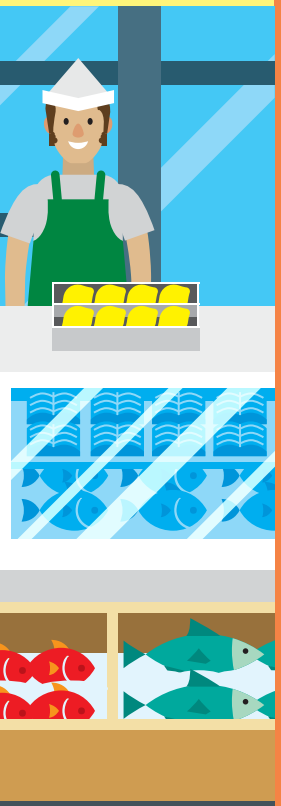
interface is a set of behaviors that a class can implement. I can see how the word can be made to apply in both situations, but it is unfortunate that the same word ends up being used in both contexts.

Do my business objects have to talk to a user interface?

No. That's the beauty of creating objects that work this way. Once you have defined channels that can be used to ask an object to do things, you can use these channels in a variety of ways. An object can be accessed by messages that originate from a network connection or by a program pretending to be a user sitting at the keyboard.

18

Advanced applications



What you will learn

You've come a long way since our early days using the Snaps framework to create egg timers and party announcers. Programs started off as something that took data in, did something with it, and then sent some more data out. That's still the case, but we now think of objects that accept messages, act on them, and pass a message on to another object. Programming isn't just about trying to work out a solution to a problem. It is as much about organizing that solution so that it is easy to test, build, deploy, and maintain.

This chapter will serve as a great sendoff for your application-development career. In this chapter, we will build on the calculator quizzer and find out how to bind data in objects to elements on a display surface. You'll also discover how programmers can create objects whose only role is to provide a view of the data in a system. This is powerful and tricky stuff. If you find yourself confused, try to remember exactly why we are performing each action and read through the "Code Analysis" sections for each sample. This material can be hard to learn, but it is knowledge that will stand you in very good stead if you ever apply for a job as a developer. To start, we will learn about some nifty C# features that you can use to speed up your program writing.

Speeding up your C#

This is a good place to mention some tricks that you can use to make writing C# code quicker and easier. You don't have to use these in all your programs, but you might find them useful and you might find them in programs that other people write.

Making statements shorter

You can use different operators to make statements shorter. So far, we have looked at operators that appear in expressions and work on two operands. Here's an example:

```
age = age + 1;
```

In this case the operator is the plus sign (+) and is operating on the variable `age` and the value 1. The purpose of this statement is to add 1 to the variable `age`. However, this is a long-winded way of expressing this, both in terms of what you have to type and what the computer will actually do when it runs the program. C# allows you to write this operation more briefly by using the line:

```
age++;
```

You can express yourself more succinctly, and the compiler can generate more efficient code because it now knows that you are adding one to a particular variable. The double plus sign (++) is called a *unary* operator because it works on just one operand. It causes the value in that operand to be increased by one. There is a corresponding `--` operator that can be used to decrease (decrement) variables.

Another example of shorthand you can use is for adding a particular value to a variable. You might make a game program where you get different scores for destroying different aliens. You could write:

```
totalScore = totalScore + alienValue;
```

This is perfectly okay, but again it's rather long-winded. C# has some additional operators that allow you to shorten this statement to:

```
totalScore += alienValue;
```

The += operator combines addition and the assignment so that the value in `totalScore` is increased by `alienValue`. **Table 18-1** shows some other shorthand operators:

TABLE 18-1 Shorthand operators

OPERATOR	EFFECT
a += b	The value in a is replaced by a + b
a -= b	The value in a is replaced by a – b
a /= b	The value in a is replaced by a / b
a *= b	The value in a is replaced by a * b

There are other combination operators as well. I'll leave it up to you to discover them. Search for "msdn C# operators" to get started.

Statements and values

In C#, statements have a value, which you can use in your program if you want to. For example, this statement assigns the value 0 to the variable `score`.

```
score = 0;
```

You might use this statement at the start of a video game to set the score value to 0. However, the statement itself has the value of 0, so if you want to, you could write this:

```
hits = score = 0;
```

This statement sets the value of the `hits` variable to the result of the statement `score = 0`. In other words, the variable `hits` is set to 0 as well. If you must do this kind of thing (and I admit I am not a fan), I'd advise you to use parentheses, like this, so that it is much clearer what is going on.

```
hits = (score = 0);
```

Using the results of unary operators in tests

When you consider operators like `++`, there is possible ambiguity in that you don't know whether you get the statement value before or after the increment. C# provides

a way of getting either value, depending on which effect you want. You can change the position of ++ to determine whether you want to see the value before or after the addition is done:

i++ Means "give me the value before the increment"

++i Means "give me the value after the increment"

As an example, the following code would make `j` equal to `3`.

```
int i = 2, j ;
j = ++i ;
```

The other special operators, += and so on, all return the value after the operator has been performed.

PROGRAMMER'S POINT

Always strive for simplicity

Don't get carried away with this. The fact that you can produce code like:

```
height = width = speed = count = size = 0 ;
```

does not mean that you should. Nowadays, when I am writing a program, my first consideration is whether the program is easy to understand. I see my duty as not to the computer, but to the next person who has to read my code. I don't think that the statement above is very easy to follow, so irrespective of how much more efficient it is, I still don't do it.

Making a Windows 10 contact editor

Let's go back to the contact application we built in Chapter 10. The lawyer you worked for has seen some other applications that she thinks are better than your program, and she is after some improvements. The first thing she would like to change is the way she finds a contact she wants to work with. At the moment, the lawyer types in the name of the contact, and the program searches for that name.

Figure 18-1 shows how this works. When the **Enter** button is pressed, the program searches for the contact with that name.

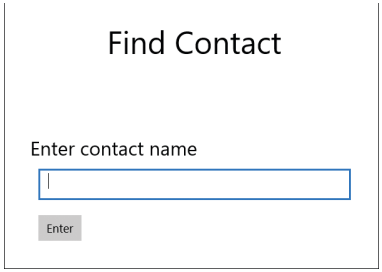


Figure 18-1 Finding contacts.

This works fine, and indeed the lawyer has been using it for a while, but now she wants something a bit easier to use. What she would like is an application in which she can pick the contact names from a list, like you see in **Figure 18-2**. The lawyer can type in a search string and then press the **Search** button. The program will then show all the contacts that contain the search string in their name, and she can select the one that she wants to work with. In the example, she has typed **r** before pressing Search. The list shows all the contacts whose name contains the letter *r*.

Figure 18-2 Improved search interface.

This seems like it might be fun to build, and we can practice our newfound XAML skills, so let's try it out.

MAKE SOMETHING HAPPEN

Make yourself a data-management application

This is a good time to start making your own full-featured, data-driven application. We are going to spend the next few pages discovering how to make a professional-grade Windows 10 contact-management application. You can use exactly the same techniques to build an application that works with any kind of data.

Storing contact details

To store the contact details in the application, we used a class named `Contact`. It held the name, address, phone number, and number of contact minutes for a given contact. This data was all held within members of the `Contact` class. Here we will use the same pattern, but we are going to make one change to the way that our "old-school" class design worked.

In the new version of the `Contact` class, all the data elements in the class will be held as public properties. We used properties in Chapter 10 when you discovered that they are a great way to get control when a program interacts with data in your object. You can refresh your understanding of properties by taking a look at the tiny `Contact` class that follows. It contains only a single member, the `Name` property of the contact. You'll see the complete class with all the data properties a little bit later.

```
public class Contact
{
    // Private string holding the name value
    private string name;

    // Public name string
    public string Name
    {
        // Get the name value
        get
        {
            // Return the value of the name
            return name;
        }

        // Set the name value
        set
```

This code runs when a program writes to the property.

```
        {
            // Set the private name to the incoming value
            name = value;
        }
    }
}
```

Once we have our class, we can use it in our programs, like this:

```
Contact rob = new Contact();
rob.Name = "Rob";
```

These two statements create a new `Contact` and then set the name of the contact to "`Rob`". When the assignment is made to the `Name` property, the `set` behavior inside the class runs and sets the value of the name to "Rob". This is a very important aspect of our new design. It means that code inside our object can gain control when something happens to the data in the object. Another important aspect of this change is that data objects can interact with the display system on the basis of properties that they expose.

In C#, you can write much shorter property definitions if all you want to do is make a value a property. These are called *auto-implemented* properties because the compiler automatically makes the private variable behind the property for you.

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

This code makes the `Name` member of the `Contact` class a property and provides `get` and `set` behaviors. However, if you want to actually get control when the property is used, you have to expand this property to the full version you saw before. Here is the complete `Contact` class that we will use for our new Time Tracker application, with all the data items as auto-implemented properties.

```
public class Contact
{
    // Data values as auto-implemented properties

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

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

    public int MinutesSpent { get; set; }

    // Constructor for a contact instance
    public Contact(string name, string address, string phone)
    {
        // Copy the incoming values into the properties
        this.Name = name;
        this.Address = address;
        this.Phone = phone;
    }
}
```

The class also has a constructor that is used to set up an instance with name, address, and phone number.

```
Contact rob = new Contact(name:"Rob", address:"Rob's house", phone: "0000 11111
2222");
```

This statement creates a new contact named `rob` from values that I entered into the program code. The completed program will read this information from `TextBox` elements.

Storing lots of contacts

The program that we created earlier held all the contact items in a `List` that was stored inside the application. This time we are going to create a class that will manage the storage. It will contain the contact `List`, provide behaviors that will let programs manage the contacts and search them, and also create a test data set for us to play with.

```
public class ContactStore
{
    // List of contacts being stored
    private List<Contact> contacts = new List<Contact>();

    // Store a contact in the store
    public void StoreContact(Contact contact)
    {
        // Add the contact to the list
```

```
        contacts.Add(contact);
    }

    // Remove a contact from the store
    public void RemoveContact(Contact contact)
    {
        // Remove the contact from the list
        contacts.Remove(contact);
    }
}
```

The preceding code shows the `ContactStore` behaviors that let a program add and remove contacts in the storage. The `StoreContact` method, shown next, is given a reference to the contact that is to be stored and adds it to the list. The `RemoveContact` method removes the given contact from the list.

```
// Create a new contact store
ContactStore store = new ContactStore();
// Create a new contact
Contact rob = new Contact(name: "Rob", address: "Rob's house", phone: "0000 11111
2222");
// Put the new contact in the store
store.StoreContact(contact: rob);
```

CODE ANALYSIS

Responsibilities in classes

Question: There are no commands in the `ContactStore` class that let a programmer edit the content of a contact. Is this right?

> **Answer:** Yes. This is all about proper allocation of responsibility. The `ContactStore` class does not have responsibility for the data in the contacts; it is the job of the `Contact` object to look after all the data relating to a contact in the program.

Question: How hard would it be to convert `ContactStore` to work with other kinds of data?

> **Answer:** Because of our sensible design, it would be very easy. All you'd have to do is change the type of the list and the parameters to the `StoreContact` and `RemoveContact` methods—the underlying behaviors would be exactly the same.

Creating test contacts

I'm very keen that we should be able to test our program without typing in lots and lots of contacts. To help, I've added a method to the `ContactStore` class that will make a contact store with a set of contacts already in it.

```
public class ContactStore
{
    // Static method that returns a test contact store
    public static ContactStore GetTestStore()
    {
        // This is the ContactStore that will hold the result
        ContactStore result = new ContactStore();

        // Array of test name strings
        string[] testNames = {
"Rob", "Mary", "David", "Jenny", "Chris"
"Simon", "Kevin", "Helen", "Neil",
"Amanda", "Sally", "Rory", "Robin" };

        // Work through each name in the list
        foreach (string name in testNames)
        {
            // Create a test contact from that name
            Contact newContact = new Contact(name: name,
                address: name + "'s house",
                phone: name + "'s phone");
            // Add the contact to the result
            result.contacts.Add(newContact);
        }

        // Return the new contact store
        return result;
    }
}
```

Finding contacts

We now have a `ContactStore` that we can use to hold contacts, but at the moment we have no way that our lawyer friend can find any of the contacts to work with.

Remember, she wants to type in part of a name–for example, **Ro**—and then have the program display a list of all the people with *Ro* in their name so that she can then pick from "Robert", "Rory", "Robin," and so on. This means that the `Find` method can't just find one contact; it actually returns a collection as the result of its search. This is something we've not done before. Up until now our methods have returned single items, but there's no reason why a method can't return a list or an array.

```
public class ContactStore
{
    // Returns a list of contacts where the name contains the
    // the search name
    public List<Contact> FindContactsWithName(string searchName)
    {
        // Convert the search name into capial letters
        searchName = searchName.ToUpper();

        // Create the list of contacts that will be returned
        List<Contact> result = new List<Contact>();

        // Loop through all the contacts in the store
        foreach (Contact contact in contacts)
        {
            // Create a capital-letter version of the contact name
            string contactName = contact.Name.ToUpper();
            // Test if the name contains the serach string
            if (contactName.Contains(searchName))
            {
                // Add the contact to the list if we have a match
                result.Add(contact);
            }
        }
        // Return the list of contacts with matching names
        return result;
    }
}
```

You can think of the `FindContactsWithName` method as a kind of filter. It takes in the list of all the contacts and builds a new list with only the matching contacts in it.

CODE ANALYSIS

The `FindContactsWithName` method

Question: What happens if the search string isn't found in the contacts?

> **Answer:** In this case the result would be a list that contains no elements, which is fine. It would mean that there is no contact in the list with the matching characters. If the search string appears in every contact's name, the result would be a copy of all the contacts in the store, which is fine, too.

Question: What is all the `ToUpper()` stuff for?

> **Answer:** We've seen this method before. The method `ToUpper` returns a version of a string as all uppercase. In other words, "Rob" would become "ROB". It is very important that the string comparisons are all performed using characters of the same case because the lawyer will complain if she searches for "Rob" and the program doesn't find "rob".

Question: Isn't building a new list each time we do a search inefficient?

> **Answer:** Not really. Remember that the list contains references to the contacts, not the contacts themselves, and references are actually very small amounts of data. The libraries underneath our programs are very good at creating lists.

Displaying a list of found contacts

The `FindContactsWithName` method returns a list of contacts for the lawyer to view. Now we need to display this list so that she can select the contact she wants to work with. To do this we can use a very powerful feature of XAML—*data binding*. Data binding does what the term implies—it makes a connection between a piece of data (in this case the list of contacts) and a display element.

Creating a data binding template

We'd like to bind a `Contact` to a display element, but we can't bind one directly because the value of a contact includes a bunch of different components. A `Contact` value contains a `Name`, an `Address`, and a `PhoneNumber` value. What we'd like to do is design a template that sets out how the contact is to be displayed. This will let us choose which parts of a contact we want to display on the screen (we might omit the phone number) and how the data will be formatted. Perhaps something like this:

```
<DataTemplate>
```

```
    <StackPanel Margin="4">
        <TextBlock Text="{Binding Name}"/>
        <TextBlock Text="{Binding Address}"/>
    </StackPanel>
</DataTemplate>
```

A data template is a lump of XAML that describes how some data is to be displayed. Here we indicate 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 the following, with the two text strings stacked:

```
Rob Miles
Rob Miles's house
```

Note that we want to display only the name and address of a `Contact` in the search list. There is no data binding for the phone number value, so it will not be shown.

Using a DataTemplate in a ListBox

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

```
<ListBox Name="ContactListBox" Margin="4" Height="300">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Margin="4">
                <TextBlock Text="{Binding Name}"/>
                <TextBlock Text="{Binding Address}"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Time to step back a little. The `ListBox` display element called `ContactListBox` will display a list of contacts that `FindContactsWithName` returns. To do this it needs to know how to lay out the display of a single `Contact`. A `ListBox` uses an `ItemTemplate` element to get the design of a list item, and in this case it will be using a `DataTemplate` to do this.

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

```
// Event handler that runs when the search button is clicked
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    // Get the search string from the search text box
    string searchName = searchTextBox.Text;

    // Get the list of matching contacts
    List<Contact> foundList = contacts.FindContactsWithName(searchName);

    // Display the found list
    ContactListBox.ItemsSource = foundList;
}

Demo 18-01 Finding Contacts
```

This is the binding point for the list.

This code runs when the user presses the **Search** button on the display. The `FindContactsWithName` method returns a list of contacts to display and then displays it in the `ContactListBox`. The `ListBox` will display each object in the collection as an element in the list. **Figure 18-3** shows how this works. The lawyer has used the search string **a**, and the program has displayed every contact with the letter *a* in his or her name.

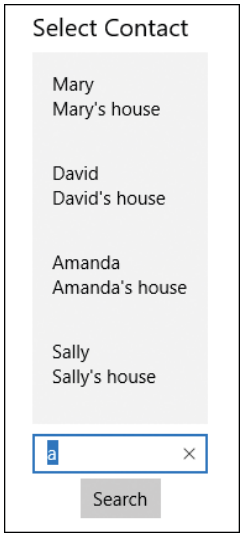


Figure 18-3 Find Filter

To generate this display, our program doesn't have to do anything other than just set the `ItemsSource` property of the `ListBox`. Everything else is done by the display. The `ListBox` even provides scroll bars automatically; if the list is too large for the screen, the user can scroll up and down within it.

CODE ANALYSIS

ListBox and ItemSource

Question: How does this work? What is Windows doing to display the items?

> **Answer:** If you run the sample code for this program, it looks a lot like magic. You type in some letters, press the **Search** button, and the contact list magically fills up with matching contacts. How does it really work? The key is the assignment:

```
ContactListBox.ItemsSource = foundList;
```

On the right of the assignment is a collection that contains references to all the matching contacts. On the left is the `ItemsSource` property of the `ListBox` that is going to show the list of contacts. The `ItemsSource` property expects to be given a collection of things to display. When the `ContactListBox` is given a collection, it works through each item in the collection, adding it to the list to display. It uses the `DataTemplate` element of the `ListBox` to determine what to display on the screen. Items in the `DataTemplate` are matched up with properties in the objects being added to the list and then displayed. In this case, the `Name` and `Address` properties are located and displayed.

What you might find confusing is that the `ContactListBox` is never explicitly told that it is displaying a `Contact` value. The `ContactListBox` is given something that happens to contain `Name` and `Address` properties, which are then displayed according to the data template in the XAML.

This is achieved by using a C# technology called *reflection*. Reflection allows a program to ask an object, "What properties do you have?" The object replies with a list of properties, and these can be matched up with the bindings in the template.

It means that the `ListBox` template we are using would work with a list that contains any elements that have `Name` and `Address` properties, not just `Contact` objects.

Question: What happens if the template refers to properties that don't exist in the list objects?

> **Answer:** The template for the `ListBox` tells it what to display for each item in the list:

```
<DataTemplate>
    <StackPanel Margin="4">
        <TextBlock Text="{Binding Name}"/>
        <TextBlock Text="{Binding Address}"/>
    </StackPanel>
</DataTemplate>
```

When the list is displayed, the reflection process matches up the template, binding names with properties in the objects. However, if we used "Binding Namey"—in other words, got the template wrong—the program would run perfectly fine; it would just fail to display anything for the `Name`. This can be a source of annoying bugs, so if something is not displayed properly, check that the template names are correct.

Question: How do I make the list look nicer?

> **Answer:** You can improve the list display by using different colors and text sizes for the name and address `TextBox` elements. You can also add any other stylistic elements you like—for example, images and shapes—to the template.

Editing a contact

We can now display a list of items really easily. The next thing we want to do is make a data-editing application. When the lawyer selects one of the items in the list, we want to display a screen that allows this item to be edited. To make this work, we have to detect when the selected element in the list changes. For this, we can use code that's very similar to the code we wrote for the `ComboBox` that's used for selecting quizzes in the previous chapter. Our program can detect when the selected item in a `ListBox` has changed by connecting some code to the event that is generated when the change happens.

The Properties page for the `ContactListBox` shown in **Figure 18-4** has the `SelectionChanged` event bound to a method that will run when the user selects one of the items in the `ListBox`.

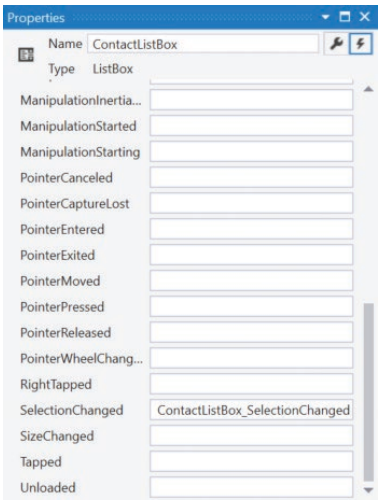


Figure 18-4 `ContactListBox` selection-changed handler.

Here is the method's code:

```
// Event handler runs when the user changes the selection in the contact list
private void ContactListBox_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    // If the change involves unselecting item, just return
    if (ContactListBox.SelectedItem == null)
        return;

    // Get the selected item as a contact
    Contact contact = (Contact) ContactListBox.SelectedItem;
    // Load the selected contact into the editor
    selectContactForEdit(contact);
}
```

CODE ANALYSIS

Selecting items

Question: Why do we have to check whether the `SelectedItem` is null?

Answer: Think about what happens when the `ListBox` is being used. There are two situations when the selected item might change. The first is the obvious one, where the user

selects an item in the list. When she does this, the event handler runs and the `SelectedItem` in the `ListBox` will refer to the item that was selected.

The second context in which the selected item might change is when the `ListBox` is loaded with a new list to display. When this happens, nothing is selected because the user hasn't chosen anything at that point. At this point, the `SelectedItem` reference is set to null, which is a value that a reference can have when it doesn't refer anywhere useful. Since a change to null is actually a change, the `SelectionChanged` method runs.

When the user selects a contact, the method `selectContactForEdit` is called by our program. The next step is to add behavior to this method that will let the lawyer edit a contact.

PROGRAMMER'S POINT

The `ListBox` selection mechanism is very powerful

You have just learned a very important principle in user-interface design. You now know how to display a list of items and get the user to select from that list. You will see this behavior in lots of applications, from social media applications as well as music players. You can use the `ListBox` to display any kind of item that you can express using XAML, and you can bind any of the properties of the items in a template to elements in your data.

The contact editor

You and the lawyer have been talking about user-interface design. Actually, talk is a polite way of putting it. The discussion has gotten heated, which is not surprising. In my experience, users have strong opinions about the way they want their programs to work, and these opinions are not always shared by the programmer. However, you have agreed on the design for the user interface shown in **Figure 18-5**.

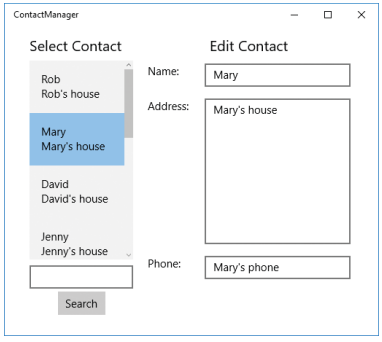


Figure 18-5 The user interface for editing a contact.

When the lawyer selects a contact, it's displayed on the right side of the page. The lawyer can edit the contact, and if she moves to a different contact, the changes are automatically saved. You have said, quite rightly in my opinion, that if the user of the program tries to move away from a contact that they have edited, the program should display a prompt such as, "Do you want to move away from this item? You have changed it. Do you want to save your changes?" The lawyer doesn't like this idea though; she says that she always gets her changes right, and she can easily put them back if she needs to. The lawyer doesn't want to be pestered by warning messages, so she has asked you not to display them.

PROGRAMMER'S POINT

The customer may not always be right, but they are always the customer

I've had discussions like this with customers. I've designed what I think are perfect interfaces, only to be told, "That's not how it should work." It hurts. There are three things I think you should remember if you find yourself in this situation.

First, never get yourself in a situation where you show the customer something that you have built and they say that they don't like it. Really important things like the user interface should be designed with the customer in the room with you. Interfaces should be the product of debate, not show and tell.

Second, from an ethical point of view, if you think something is wrong in a dangerous way, you should point it out to the customer and formally request them to take responsibility for the behavior. This program doesn't fall into this category, but if you were writing an application with a medical purpose, you really must point out any design errors that you think might result in invalid data being displayed.

Finally, even if the customer is wrong, they are still the customer. They are paying you to make something that they want. At the end of the day, you want them to be happy with what you have made. And sometimes, in my experience, they may even be right.

When you select a contact, the edit pane is filled with the contact's details, which you can use or edit. Here is the code for the `selectContactForEdit` method:

```
// This is the currently selected contact, initially null
Contact selectedContact = null;

// Selects a particular contact for editing
private void selectContactForEdit(Contact contactToEdit)
{
    // Check to see if we are currently editing a contact
```

Selects a different contact for editing.

```
    if (selectedContact != null)
    {
        // We are about to move off a contact - save it
        saveContactFromPage(selectedContact);
    }
    // Display the contact that we are moving to
    placeContactOnPage(contactToEdit);
    // Remember the selected contact so it can be saved later
    selectedContact = contactToEdit;
}
```

It's important to understand the context in which the `selectContactForEdit` method runs:

1. The user searches for a contact (types **Ro** in the Search text box, for example, and presses the **Search** button). The list fills with contacts that have the search string in their name.
2. The user selects a contact in the `ListBox` (clicks on Rob in the list of contacts that are displayed).
3. The `SelectionChanged` method runs because it is connected to the selection-changed event in the `ListBox`, and we have just selected a different contact.
4. The `selectionChanged` method then calls `selectContactForEdit` to get ready for editing.

The `selectContactForEdit` method has to do two jobs. If must save the contact currently being edited (if there is one), and it must then select the new contact for editing. The variable `selectedContact` is used by the program to keep track of the contact currently being edited. When the program starts, this variable is set to null, to indicate that no contact is being edited.

Next, here are the methods that move data between a contact and the `TextBox` elements on the page that the user sees. Each method is given a contact as a parameter and either moves contact information from the contact onto the screen (`placeContactOnPage`) or moves contact information from the page back into the contact (`saveContactFromPage`).

```
// Place a contact on the page and make it ready for editing
private void placeContactOnPage(Contact source)
{
    // Copy the data from the source contact into the display elements
```

```
    NameTextBox.Text = source.Name;
    AddressTextBox.Text = source.Address;
    PhoneTextBox.Text = source.Phone;
}

// Load contact data from the display elements into the destinatioon contact
private void saveContactFromPage(Contact destination)
{
    // Copy the display element into the destination contact
    destination.Name = NameTextBox.Text;
    destination.Address = AddressTextBox.Text;
    destination.Phone = PhoneTextBox.Text;
}

Demo 18-02 Contact Editor
```

CODE ANALYSIS

Editing contacts

Question: Is there any data validation in this process?

Answer: The two methods, `placeContactOnPage` and `saveContactFromPage`, don't validate any data. If the lawyer leaves any of the `TextBoxes` empty or enters invalid information, the program will not display any errors. The lawyer is fine with this, but if your application needs to ensure that the data held is valid, it would be sensible to put some validation in here.

Data binding in the user interface

If you run the contact editor that we have made so far, you'll find that it works quite well. You can select a contact, edit it, move to another contact, and then move back to the original. You will find that your edits are being saved on the contacts, but the user interface is not as good as it could be. **Figure 18-6** shows the problem.

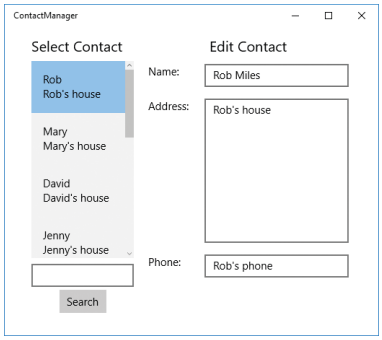


Figure 18-6 Displaying updates.

Here, I've changed the name of the first contact to Rob Miles, but the entry in the selection list on the left hasn't changed to reflect this. We are fairly sure that our lawyer customer will spot this and won't like it. So we need to fix it.

Making objects observable

To do this, we have to make the display system "aware" of changes in data that needs to be updated. The display system for the list needs to be informed that a name has changed so that it can update the `ListBox`. In XAML, we do this by making objects "observable."

In the context of XAML, observable means, "I can ask you to tell me when you have changed." Most C# objects are not observable because nothing wants to know whether their value changes. The list of contacts being displayed by the `ContactListBox` is not observable. This is fine if all we want to do is view the content of a list, but as we have seen, if any elements in the list change, there is no way the display can discover this and update. What we need is a more specialized form of collection that can be "observed" by the display system so that changes to the collection can be reflected in the display.

The `ObservableCollection` class

This is what the `ObservableCollection` class 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 display system can be notified of this. We can create an `ObservableCollection` of contacts very easily:

```
// Observable version of the contact list
```

```
ObservableCollection<Contact> contactList;

// Event handler for the search button
private void SearchButton_Click(object sender, RoutedEventArgs e)
{
    // Get the name to search for from the search textbox
    string searchName = searchTextBox.Text;

    // Get the list of matching contacts
    List<Contact> foundList = contacts.FindContactsWithName(searchName);

    // Use the found list to create an observable collection
    contactList = new ObservableCollection<Contact>(foundList);

    // Connect the collection to the contct listbox to display it
    ContactListBox.ItemsSource = contactList;
    // Clear the edit display
    clearContactEdit();
}
```

CODE ANALYSIS

ObservableCollection

Question: What is the difference between `ObservableCollection` and `List`?

Answer: From a user's point of view, there is no difference between the two collection classes. They work in the same way. The difference is that the Windows management system can connect to events that `ObservableCollection` generates if the contents of the list change. The clue is in the name. One class can be watched, the other can't.

Question: Why don't we make all the collections observable?

Answer: We could use `ObservableCollection` for all our data storage, but this might slow down our programs. An observable collection has to check to see whether anyone is interested in any changes that it is making to the data it's managing. This slightly slows down the operation of the collection, and it's why the type is reserved for items that we are displaying.

Unfortunately, if we add the new version of the **Search** button event handler, shown in the preceding code, the program still doesn't reflect changes we want. If the user updates a name in a contact, this will not be reflected by the text in the list. This

happens because `ObservableCollection` responds to the changes in the contents of the list, not to changes to the data in an element in the list. If we added or deleted a contact, the list would change, but just changing the content of an element in the list is not detected. In other words, the list currently has no way of knowing when the data in it has changed.

Making observable contacts

Each of the XAML display elements that make up the Time Tracker user interface is displaying a view of a data object in our program. The editor has created that view by setting values on properties in the display elements.

The `placeContactOnPage` method shown next takes the name, address, and phone information from the contact referred to by `source` and sets the `Text` properties of the appropriate `TextBox` values so that the contact details are displayed. The snag is that this is a one-time operation. If anything in the `Contact` changes, the display will not be updated because at the moment the `Contact` class is not observable. We have to give the `Contact` class some behaviors that allow other objects (in this case, the display system) to be notified when properties in a `Contact` are changed.

```
// Place the contact on the page
private void placeContactOnPage(Contact source)
{
    // Copy the contact information from the contact onto
    // display components
    NameTextBox.Text = source.Name;
    AddressTextBox.Text = source.Address;
    PhoneTextBox.Text = source.Phone;
}
```

The XAML environment has a protocol by which this notification is performed. It is called `INotifyPropertyChanged`. With `INotifyPropertyChanged`, an object contains a "list of people to tell" if something changes in the object. It's a little like when you are planning a party. Somewhere you'll have a list of folks you've invited. If you need to change the venue or the starting time (in other words, if any of the "properties" of your party changes), you will run through the invitation list, call the people on it, and tell them that, for example, the party now starts at 8:00 not 8:30.

The "list of people to tell" in our object is a list of *delegate* objects. A delegate is a C# feature we've not seen before. Delegates are extremely powerful. You can regard a delegate as an object that contains a reference to a method in an object. If the delegate is "called," the method that the delegate refers to will be called.

CODE ANALYSIS

Delegates

Question: Where have we seen delegates already?

> **Answer:** We've seen delegates in action, but we've not created them ourselves. Delegates provide the mechanism by which a XAML button is connected to a method in our program. The **Search** button in the Time Tracker program "knows" which method to call when it is clicked because it has been given a delegate that refers to that method. We have discussed the way that an event is just a method call—a delegate allows one object (the receiver) to hand a delegate to another (the transmitter) so that if the event happens, the transmitter can call that method.

You can think of a delegate as something like a "business card" for a method. If another object has the business card, it can call the method. When the Time Tracker application starts, the XAML system creates a delegate for the `SearchButton_Click` method and then hands it to the `SearchButton` object. This is how our method gets called when the button is clicked. When property change notifications are being used, the display system gives a "business card" to the data item so that the data item knows whom to call if the data in it changes.

Delegates provide a way that objects can bind themselves together and create paths for messages when the program runs.

Here is what the "list of people to tell" looks like in an object that wants to become observable. The `event` type is a `delegate` type that is built into C# specifically for event handling. Anything that is interested in changes to an object—that is, an object that wants to observe me—can attach a delegate to the `PropertyChanged` item in the class. We actually never assign anything to this variable, but the display system will.

```
public event PropertyChangedEventHandler PropertyChanged;
```

Now that you know how to keep a "list of people to tell," we have to discover how to actually tell someone that the value has changed. We do this by calling the delegate:

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

A method that is connected to the `PropertyChanged` delegate accepts two arguments. This information is fed into the display system to tell the display that something has changed and needs to be redrawn. The first item of data is a reference to the actual object that contains the property that is changing. The second is a `PropertyChangedEventArgs` value, which is set with the name of the property that is changing.

The code above tells an observer that the `Address` property of the contact is being changed.

We want this code to run every time the address in a contact changes, so we put it in the `set` behavior of the address property.

```
// Contact class that implements the INotifyPropertyChanged methods
public class Contact :INotifyPropertyChanged
{
    // Private data value for the address
    private string address;

    // Binding point for objects that want to receive
    // notifications when this property changes
    public event PropertyChangedEventHandler PropertyChanged;

    // Public address property
    public string Address
    {

        // Read the property
        get
        {
            // Just return the address value if the property is read
            return address;
        }

        // Set the property
        set
        {
            // Store the incoming address value
            address = value;
            // Test if anything has connected to the
            // property changed event
            if(PropertyChanged != null)
            {
                // Fire the property changed event
                PropertyChanged(this, new PropertyChangedEventArgs("Address"));
            }
        }
    }
}

Demo 18-03 Updating Contact Editor
```

CODE ANALYSIS

INotifyPropertyChanged

This is probably the most complicated piece of C# you have seen so far. And, of course, you have some questions.

Question: Tell me again why are we doing this?

> **Answer:** This code provides a way to tell the display system when data in our object changes. The display system needs to know when a value changes so that it can update the view of the data. This is extremely powerful. It means that if our program makes a change to a contact being displayed—for example, by assigning a new address to the contact—the display will update automatically, without us having to do anything else.

Question: Why does `PropertyChanged` not provide the new value of the property?

> **Answer:** It seems a bit silly for the `PropertyChanged` method not to provide the new value of the property. It is a bit like me calling you and saying, "The time of the party has changed" and then ending the call. Of course, if I did that you'd instantly call me back and ask me, "Okay, what's the new time then?" That's what's supposed to happen in the case of the display system. The `PropertyChanged` event doesn't deliver the new value; it triggers the display system to fetch the updated value and display it at some time in the future.

Question: What happens if you get the name of the property wrong?

> **Answer:** If the program uses `Adress` rather than `Address` in the parameters to the `PropertyChanged` event, the program would run without an error, but the update would not take place.

Question: How can a class implement an interface that doesn't seem to contain any methods?

> **Answer:** The `Contact` class must implement the `INotifyPropertyChanged` interface so that the display system knows it can be observed. An interface is a "shopping list" of methods. Any class that implements the interface must contain the methods that the interface describes—except the `Contact` class doesn't seem to implement any extra method, just the `PropertyChanged` event.

An interface can contain events as well as methods, which makes really good sense. If you use interfaces to describe how objects can be connected, it should be possible to use event delegates to do this.

If we make all the properties perform the appropriate notification, we get an edit application that actually updates correctly, as you can see in **Figure 18-7**.

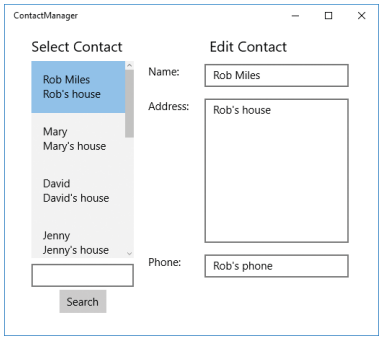


Figure 18-7 Correct updates.

If you run and work with the demo application in **Demo 18-03 Updating Contact Editor**, you'll note that the text in the `ListBox` doesn't update until the user moves to a different item in the list. This is because the content of the `Contact` is only updated at that point, and that is the action that triggers the update of the `ListBox` contents. It is possible to make a version that updates the list every time the text in the contact changes, but I think this would be rather distracting to use.

CODE ANALYSIS

Observability

Question: Why did we have to make the list an `ObservableCollection` if it is actually changes in the data in the list that need to be displayed?

> **Answer:** The list has to be observable because the bindings for "observability" are passed down from the container to the objects that it contains. The display system is bound to the `ListBox`. But when contacts are added to the `ListBox`, each of them is bound to the `ObservableCollection` in the `ListBox` so that changes are propagated correctly. You can think of this as a "chain of command" that has to be connected continuously from the top to the bottom.

Software design and the Time Tracker

Your younger brother is quite impressed with the Time Tracker program and has plans to convert it into a "cupcake recipe tracker" for his cupcake-manufacturing friend. But

he's also been reading some software development articles online and has decided that the Time Tracker is not very good because it "doesn't use Model-View-View-Model." You suspect that he actually has no idea what this means, but we might as well take a look at the technology since he mentioned it. It turns out that he is right, and there are a number of reasons why Model-View-ViewModel (or MVVM) is worth investigating.

Model-View-ViewModel

A program with a graphical user interface can be very hard to test. The only way that we can discover whether everything on the display works is to type things in, press the buttons, and watch what happens. You might get your programs tested once or twice by paying your younger brother to run through a test sequence, but if you want to test your program frequently, this could get very expensive. Testing user interfaces is a big problem for software developers because they really want their tests to be completely automatic. The reason our contact editor is hard to test is that the editing behaviors are mixed in with the display elements on the page, and there is no separation between the display and the processes behind it. To test the system we have to write code that talks to lots of different elements in the program.

PROGRAMMER'S POINT

Learn about design patterns

A design pattern is a way of structuring a solution. Every profession has its own design patterns that are learned from experience. For example, if you are painting a room, one "pattern" is to paint the walls first and then paint the door and window frames last. Design patterns have been created based on the experience of many developers and projects. From a professional perspective, you should investigate design patterns as you build your programming skills. Model-View-ViewModel is not the only pattern that you will need to have some knowledge of.

Oh, and one word of warning. Just like you can find decorators who swear that you should paint the door and window frames first, you will find developers with different opinions about the different patterns. The best advice I can think of given this situation is to build up your experience and knowledge so that you can find your own voice on the subject. And remember that your opinion is at least as valuable as that of other people.

Our program would be much easier to test if we had a single object that looked after just the editing process. Model-View-ViewModel has been designed to provide this. A solution that has been structured using the MVVM pattern has a view-model class that provides the link between the XAML, which provides the view of the data, and the `Contact` class (in our program), which provides the model of the data that we are working with.

You can think of a view-model class as a bit like a waiter in a very posh restaurant. The diner (the view) is sitting at a table selecting items to eat, and the cook (the model) is in the kitchen preparing the food. The diner will ask the waiter (the view model) for items from the menu, and the waiter will create requests to the cook for meals. When the meal is ready, the waiter takes it from the kitchen and delivers it to the diner. The diner has no idea how the food is prepared; she just knows what she wants to eat. The waiter has knowledge of the commands that the cook understands and will translate the diner's requests as required.

There are a lot of advantages to this structure. Different diners (views) can order meals. The cook can be replaced with a different cook, and as long as the new cook and the waiter use the same commands to communicate, the restaurant will continue to function. The huge benefit from a programmer's pointer of view is that one of our views can be a "test" view that makes requests and can then check the response from the view model.

Figure 18-8 shows how the objects communicate. The connection between the view and the view model (diner and waiter) is achieved by data binding. The view is a page of XAML, and the view model contains all the code that actually drives the user interface. The job of the model is to provide access to the data that the system is managing.

Figure 18-8 Model-View-ViewModel pattern.

Data binding in Model-View-ViewModel

In the earlier versions of the XAML contact manager, our program explicitly moved data in and out of the display elements for editing. But there is a much easier way of doing this, which is to use XAML data binding to connect properties in the view-model class to elements on the screen.

```
<TextBox Name="NameTextBox" Width="200" Margin="4"
Text="{Binding Name,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"></TextBox>
```

This XAML describes a data-bound `TextBox` called `NameTextBox`, in which the text in the box is bound to a `Name` property in the view-model class. That sounded pretty sophisticated, didn't it? Let's take it again in slow motion. We can start with the way that we have performed editing up until now.

```
<TextBox Name="NameTextBox" Width="200" Margin="4" Text="Robert"></TextBox>
```

The XAML above defines a `TextBox` named `NameTextBox`, which contains the fixed text "Robert". When the program runs, it displays "Robert" in the `TextBox`. This is a great way to display my name, but not what we want to do. We want to display the name of the contact that the lawyer is working with. We can work with the name of a contact by loading it into this `TextBox`:

```
private void placeContactOnPage(Contact source)
{
    NameTextBox.Text = source.Name;
}
```

We use the `placeContactOnPage` method to make a contact visible for editing. You can see a shortened version of it just above. The statement in the method puts the name of the source contact into the `NameTextBox` so that the user can see and edit the name. At the end of the editing, the program must put the edited text back so that any changes that are made are reflected in the contact being edited.

```
private void saveContactFromPage(Contact destination)
{
    destination.Name = NameTextBox.Text;
}
Demo 18-02 Contact Editor
```

We call the `saveContactFromPage` method when we complete an edit. This shortened version contains a statement that puts the edited name back into a destination `Contact` at the end of editing. You can see the code in action in the **Demo 18-02 Contact Editor**.

We have to do this for all the items being edited. But we can remove the need for this code by using data binding in the XAML. We specify the source of a XAML property as being bound to a property in the view-model class.

This is a data-bound version of the `NameTextBox`:

```
<TextBox Name="NameTextBox" Width="200" Margin="4"
Text="{Binding Name,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"></TextBox>
```

Everything about the TextBox is the same, except that the text for the name comes from a data-bound property in the view-model class. We will see this Name property in the next section.

Let's go through the details of the binding, looking at each item in turn.

```
Text="{Binding Name,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"
```

The first item, the Binding, identifies the property in the class we are binding to. We could also have a binding to an Address property for the TextBox that lets the user edit the address of a contact.

The second item, Mode, specifies the *mode* of the binding. The mode can be OneTime, OneWay, or TwoWay. If the mode is OneTime, the source data is read once when TextBox is initially displayed. If the data in the TextBox is updated by the program, this will not be reflected on the screen.

A OneWay binding means that the XAML control will change if the property changes (in other words, if the program changes the content of the Name property, the display will change to match), but if the user types something into the TextBox, this change is not reflected in the view-model class. The mode that we want to use is TwoWay. This means that if our program changes the data, the display updates, and if the user changes the displayed name, the data is updated as well.

The final item specifies when the source item (the property in our view model) is updated. I think this is a very badly named item. It makes you think that you are defining something that will update the source trigger. In fact, what the statement means is, "This is the trigger that will cause the source property to be updated." In the case of our editor, we want the display manager to update the information in our view-model class whenever the property changes (that is, when the user types something into the TextBox or when the program changes the property), so we use the PropertyChanged setting.

CODE ANALYSIS

Making sense of data binding

Question: Data binding looks complicated. Can you remind me again why we are doing it?

Answer: We implement data binding because we want the view element of our design to contain no program code at all. We want changes in the display to be reflected in properties in our view model without us having to do anything. In other words, the value of the Name property inside our view-model class should be automatically visible to the user in

the display, and changes that are made in the display should be reflected in the content of the property.

Binding is a very good term for what is happening. When two items are bound, changes in one object are automatically reflected in the other without us doing anything, which is a good thing.

Question: How does data binding actually work?

Answer: You can think of data binding as a set of instructions to the display environment that sets up a connection between two objects. We have already seen a fundamental element of the process in the `INotifyPropertyChanged` mechanism. This provides a way that one object can say to another, "Hey! I've changed!", which can then use that information to update something in the system. Data binding uses `INotifyPropertyChanged` to cause changes in one item to propagate through objects in a system.

Question: Can you only bind to text properties?

Answer: No. In fact, this is one of the most powerful aspects of data binding. You can bind program properties to lots of other properties of elements on the display, including their position, their color, or even their size. The process is exactly the same..

The view-model class

The view-model class manages the interaction between the view and the data objects (otherwise known as the model). The property that the XAML is connected to is defined in our view-model class, and the view model also contains all the behaviors that make the editing work.

```
public class ContactManagerViewModel : INotifyPropertyChanged
{
    // Private name string
    private string name;

    // Public name property
    public string Name
    {
        get
        {
            // Return the private value for a get
            return name;
        }
        set
```

```
        {
            // Set the private property to the incoming value
            name = value;

            if (PropertyChanged != null)
            {
                // If an object has registered an interest in the property,
                // call the PropertyChanged event to indicate the name has changed
                PropertyChanged(this,
                        new PropertyChangedEventArgs("Name"));
            }
        }
    }
}
```

The preceding code shows the `Name` property in the view-model class. If you think you have seen this arrangement before, you have. We have the same code inside the `Contact` object so that the display system can bind to the contact and detect when the name of the contact changes. The actual view model in our application will have properties like this for each data item that the view model is presenting to the view. In our case, that is the `Address`, the `Phone` number, the search `ListBox`, and the search text.

Connecting the view model to the view

We now have a XAML view and a C# view model. The next step is to connect the two. We do this by setting the *DataContext* for the edit page to the view-model class. The data context defines the object into which all the bindings will be mapped. In the case of our editor, we want this context to be an instance of the `ContactManagerViewModel` class.

```
<Page
    x:Class="ContactManager.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ContactManager"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Page.DataContext>
        <local:ContactManagerViewModel x:Name="contactManagerViewModel"/>
    </Page.DataContext>
```

Standard XAML header for the editor page.

Start of the DataContext information for the page.

Identifies the class to use and the local name.

The `local:ContactManagerViewModel` part of the description identifies the class that is to be created. The `x:Name="contactManagerViewModel"` part of the description names the instance that is created.

CODE ANALYSIS

Using a DataContext

Question: What is a `DataContext` exactly?

> **Answer:** The XAML bindings contain the names of items that we want to bind to. For example, we are binding the `Name` of our contact to a `TextBox` on the display. The `DataContext` is the link between the XAML and the C# object that actually contains a `Name` property that holds the value that will be bound. It is the "context in which the data exists." We could use any class we like as the `DataContext` for the XAML page as long as the class contains properties that match up with the ones on the page.

Question: How is a `DataContext` used when the program runs?

> **Answer:** When the page is loaded, the system creates an instance of the view-model class. In this case, it creates an instance of the `ContactManagerViewModel` class, which will be called `contactManagerViewModel`. We've seen this kind of object creation before. Each of the display elements in the XMAL is also created when the page is loaded.

Question: Where does the connection to the data get set up?

> **Answer:** The view-model class has the job of connecting the view to the data. This happens when the view-model instance is created. This is when the application would load the data from whatever storage is being used and then make it ready for use by the editor.

Passing commands to the view model

Data binding lets us connect properties in the view-model class to display elements in the view. But it doesn't provide a means by which we can pass command actions into the view model. We need to get one command into our view model, and that is the one that initiates a search of the stored contacts to find contacts with a name that contains a particular string. In the earlier application, we created an event handler that runs when the **Search** button is pressed. We can do the same thing in our view-model application, except that this time the event handler just calls a method in the view model:

```
namespace ContactManager
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        // Constructor for the page
        public MainPage()
        {
            this.InitializeComponent();
        }

        // Search behavior bound to the button
        private void SearchButton_Click(object sender, RoutedEventArgs e)
        {
            // Ask the view model to perform the search
            contactManagerViewModel.DoSearch();
        }
      }
}
```

When the user presses the **Search** button, the event handler makes a call to the `DoSearch` method inside the view-model class.

```
public class ContactManagerViewModel : INotifyPropertyChanged
{
    public void DoSearch()
    {
        List<Contact> foundList = contacts.FindContactsWithName(SearchText);

        FoundList = new ObservableCollection<Contact>(foundList);
    }
}
```

This is the entire content of the `DoSearch` method in the view model. It asks the model (the contacts store) to produce a `List` of all the contacts that contain the search text in their name. Then it makes an `ObservableCollection` from this list and sets the `FoundList` property to the list it has found. What the code does is exactly the same as in the previous version of the editor, but it is much, much, simpler.

CODE ANALYSIS

Model-View-ViewModel Magic

Question: How does the event handler find the view-model class to call?

> **Answer:** When we set the `DataContext` in the XAML, we say, in effect, "All the data bindings in this design are bound to properties in a class named `ContactManagerViewModel`." The instance of that class that you are going to create is called `contactManagerViewModel`. This instance is created when the page loads and can then be used in code that runs inside the view class.

Question: How does the `DoSearch` method get hold of the string to search for?

> **Answer:** In the previous version, the program had to go and find the `TextBox` that contains the part of the name to search for. In this case, we just use the property `SearchText`. This property is bound to the view and delivers the text that we want to search for. The view model doesn't know where the binding takes place, or even what control the property is bound to; it just knows that this property will hold the text to search for. This illustrates the magic of MVVM.

Question: How do we set the contents of the `ListBox` to the results of the search?

> **Answer:** This is more view-model magic. The `FoundList` property in the view model is bound to the `ItemsSource` property of the `ListBox` that displays the list of contacts. When the assignment is performed by the second statement in the `DoSearch` method, the `ListBox` automatically updates.

If you learn more about the Model-View-ViewModel pattern, you will find a way of using a command mechanism that allows events to be directly bound to methods in the view-model class. However, for our purposes, this design is fine.

Detecting changes in the view

The way the program works, the lawyer will select contacts from the `ListView` when she wants to work with them. We can use data binding to detect when the selected contact changes.

```
<ListBox ItemsSource="{Binding FoundList}" Name="ContactListBox" Margin="4"
Height="273" SelectedItem="{Binding SelectedContact,Mode=TwoWay}" >
```

When the selected item changes—in other words, when the lawyer selects an item in the `ListBox`—the selected item will change. The binding will cause the `SelectedContact` property in the view-model class to change.

```
public class ContactManagerViewModel : INotifyPropertyChanged
{
    private Contact selectedContact;

    public Contact SelectedContact
    {
        get
        {
            return selectedContact;
        }
        set
        {
            if (selectedContact != null)
                saveContactFromPage (selectedContact);

            selectedContact = value;

            if (selectedContact != null)
            {
                placeContactOnPage(SelectedContact);
            }

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

Demo 18-04 MVM ContactEditor
```

The `set` behavior in a property lets a program get control when a property is changed. When a contact is selected, we want to save the contact we were previously editing and then select the new one for editing. That is what the preceding code does. If you run the demo program, you will find that it works exactly as it should, but all the behaviors are managed in the view-model class.

PROGRAMMER'S POINT

Some things you learn by studying code

The Model-View-ViewModel principle is an important one when creating programs. However, it is not something that will come to you just by reading descriptions of how it works. The only way that you can really understand how it works is by considering what the programmer is trying to do and then looking at the code that actually does it.

I have learned some great programming lessons from studying code written by other people. I go from, "Why on earth did they do that?" to "Blimey, that is rather clever." If the past few sections have been hard to get your head around (and I would quite understand that), you can learn a lot from studying the code.

Remember that you don't just have to look at the text though. You can perform "experiments" on it by inserting break points using Visual Studio and then stepping through the code as it runs.

MAKE SOMETHING HAPPEN

Investigate MVVM

Now that you are starting to understand how MVVM works, take some time to build something simple using the pattern. You can take an existing program and try to work out what would go into the view, what would be in the view model, and what the model itself should contain. You could also build an experimental application that links an input `TextBox` to an output `TextBlock` by just using data binding.

Remember that you can use data binding to control every aspect of items on a XAML page, including images and sounds.

What you have learned

You have spent a lot of time in this chapter learning about the Model-View-View-Model design pattern, which is used to build user interfaces that are easy to test and manage. You have found out about *patterns,* which set out a way of doing things. You have also discovered the principle of data binding, through which, behind the scenes, code can be used to enable the changes in one object to cause changes in another. You have seen how properties are an important part of data binding, as they allow

us to trigger code actions when data in a class is changed. You have also investigated *delegates*, objects that allow a program to manipulate references to methods.

This has been a great chapter for exercising your skills with objects, and for learning the importance of design. And, of course, here are some questions.

Is Model-View-ViewModel the only way to structure applications?

No. There are lots of ways to create an application with a user interface. The Model-View-ViewModel design was created with the XAML page-description language in mind. The principle of data binding is used in many different applications; knowing how it works is very useful. However, if you want to create all your applications by using the "code in the page" techniques that you saw in the previous chapter, that is fine by me, and it will be fine by your users, too.

How do we actually test applications built using view models?

Once you have your view-model class, you can test this as you treat any other software object. You can trigger behaviors on the object and compare what happens with what should happen. For example, you can test the search behavior of our editor by creating a test data set, setting some text in the `SearchText` property, calling the `DoSearch` method, and then looking at the content of the `FoundList` collection. The beauty of this is that at no point does anyone have to look at the screen to test, and there is no chance of there being any bugs in the code for the view because the view does not contain any code at all.

Can a program contain more than one view model?

Yes it can. A given view model is usually associated with a particular activity in an application. In a banking application, you might have a different view model for each type of transaction that the system will perform.

Why are delegates so useful?

You can regard a delegate as a reference that can refer to methods. You have seen them used to implement buttons in a user interface. The `Button` object is given a delegate object that identifies a method to call when the button is activated. The `INotifyPropertyChanged` mechanism uses delegates, too; they are how an object can register an interest in changes to a property. The observing object gives the property a delegate that can be called if a change occurs. However, the power of delegates extends beyond these two applications.

You can turn a list of delegate references into a "program inside a program." Each delegate could be called in turn to provide a sequence of actions. I've used this to very good effect in games to provide "scripts" for my game elements to follow. Understanding how delegates work and how to use them is a really powerful skill.